HIGH-PERFORMANCE EXPLICIT TRANSIENT STRUCTURAL ANALYSIS

THÈSE N° 2200 (2000)

PRÉSENTÉE AU DÉPARTEMENT DE GÉNIE CIVIL

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

PAR

Pieter Theodorus Gerardus VOLGERS

M.Sc. in Aerospace Engineering, Delft University of Technology, Pays-Bas
de nationalité néerlandaise

acceptée sur proposition du jury:

Prof. L. Pflug, directeur de thèse
Prof. A. Curnier, rapporteur
Dr R. Gruber, rapporteur
Dr S. Merazzi, rapporteur
Dr C. Rankin, rapporteur
Prof. E. Riks, rapporteur

Lausanne, EPFL
2000

Abstract

The continuous development of finite element programs creates code which is increasingly difficult to maintain, due to the increasing complexity as well as the chosen programming language. At the same time, the desire to analyse increasingly more complex and larger models requires the use of powerful tools capable of solving these problems in a reasonable amount of time. The research described in this thesis aims to solve these issues of maintainability and performance. For that purpose a new code structure has been designed, and the new finite element program has been adapted to make use of modern high-performance computers. The two principle parts in this thesis are:

Design of an object-oriented code base

In order to improve maintainability and ease of development, an object-oriented framework for finite element programs, written in C++, has been designed. The structure of the framework is set up in such a way, that the existing computational routines from the old Fortran program can be integrated in the new object-oriented code.

Adaptation for high-performance computing

Although object-oriented code generates more overhead than traditional procedural programming, the restructuring of the program together with

the use of certain advantages of C++ created a new program executable which is faster than the old program. The object-oriented code made the adaptation of the code for parallel computing relatively easy and allowed the programming of the communication to be implemented transparently.

New element types and new material models have been implemented to demonstrate the flexibility and ease of development of the new code. Three applications are presented to demonstrate the possibilities of the new finite element program.

Version abrégée

Les programmes de simulation numérique basés sur la méthode des éléments finis ou volumes finis sont de plus en plus difficiles à maintenir, en raison notamment de la taille croissante des programmes et du choix des langages de programmation. Parallèlement le besoin de résoudre des problèmes de plus en plus complexes requiert des outils capables de traiter ces applications dans un temps raisonnable. Le présent travail consiste à trouver des solutions à cette évolution dans le domaine de la performance et de la maintenance. A cette fin une structure spécifique de code de simulation a été élaborée et le programme qui en résulte a été particulièrement adapté aux calculateurs parallèles modernes. Cette contribution est composée de deux parties:

Conception d'une base de programmes orientés objet

Afin de simplifier le développement des codes éléments finis et d'améliorer leur maintenance une base de programmes orientés objet a été développée en C++. La structure a été conçue de manière à pouvoir intégrer les programmes existants codés dans les langages traditionnels tels que C ou Fortran.

Calcul à haute performance

Bien que les programmes codés dans un langage orienté objet tel que C++ provoquent une consommation de ressources plus substantielle que les programmes codés dans un langage traditionnel tel que Fortran, la restructuration, combinée à l'exploitation de certains avantages propres au C++, produit un code plus efficace. En plus le modèle de programmation orienté objet facilite considérablement l'implémentation sur ordinateur parallèle, rendant transparent l'effet de la communication entre processeurs.

Des nouveaux types d'éléments et des nouveaux modèles de matériaux ont été introduits afin de montrer la flexibilité de l'approche et la facilité avec laquelle ces théories peuvent être implémentées. Trois applications sont présentées, elles démontrent les possibilités du code.

Samenvatting

De voortdurende ontwikkeling van de huidige eindige elementen programma's resulteert in code die steeds moeilijker is te onderhouden. De wens om ingewikkelde en grotere modellen te bestuderen vraagt tegelijkertijd om de middelen om deze problemen in een redelijke tijd op te kunnen lossen. Het onderzoek in dit proefschrift is bedoeld om aan de problemen van steeds intensiever onderhoud en de hoge eisen ten aanzien van de prestaties het hoofd te bieden.. Hiervoor is een nieuwe programma structuur ontwikkeld en het nieuwe eindige elementen programma is aangepast voor het gebruik van moderne supercomputers. De twee belangrijkste onderdelen in dit proefschrift zijn daarom:

Ontwerp van een object-georienteerde programma structuur

Ter verbetering van de eenvoud van onderhoud en ontwikkeling is een object-georienteerde structuur voor eindige elementen programma's, geschreven in C++, ontwikkeld. Deze structuur is dusdanig opgezet dat de bestaande reken-routines van de oude Fortran code in de nieuwe structuur kunnen worden opgenomen.

Aanpassing voor supercomputers

Ondanks het feit dat object-georienteerde code in meer overhead resulteert dan in traditionele procedurele programma's, de herstructurering

van het programma en het gebruik van de mogelijkheden van C++ resulteert in een programma wat sneller is dan de oude versie. De object-georienteerde opzet maakte de aanpassing van de code voor parallelle computers relatief eenvoudig en maakte het mogelijk de communicatie doorzichtig te programmeren.

Nieuwe element types en nieuwe materiaal modellen zijn geschreven om de flexibiliteit en de eenvoud van ontwikkeling in de nieuwe code aan te tonen. Drie toepassingen laten de mogelijkheden van het nieuwe eindige elementen programma zien.

Contents

List of Figures

List of Tables

1

Introduction

During the last fifty years the development of digital computers have opened up the use of numerical analysis of ever more complex systems. The study into the behaviour of a continuous medium, previously being limited to relatively simple structures and analytical solutions, has seen an impressive development in the ability to analyse ever more complex problems. In computational structural mechanics, the finite element method has become the standard analysis tool for many engineers and is still under constant development.

1.1 Overview of today's problems

The development of new elements, advanced material models and so forth, is generally done in long existing codes, originally written 20 or 30 years ago. They were written according to the standard of the time of sequential, or procedural, programming, usually Fortran. Computer science has not stood still over this period, and new programming methods and languages have come to the scene. Particularly the object-oriented programming model has been designed to solve some of the complica-

tions which come with increasingly large and complex programs.This programming model greatly improves program design, development and maintenance, by putting different parts of the code in so-called objects and protecting them and their content from the rest of the code. This is the fundamental difference from sequential programming, where all data and code are put together and allocated at the base level of the program. For large programs this results in 'spaghetti-code', where everything is entangled.

The problem with the development of existing finite element programs is the (fundamental) setup of the code. During development an increasing amount of time is now spent on integrating the new parts into the existing code and maintaining it all. Errors can easily be introduced which have their effect on other parts of the program or when solving other problems than the one the developer is working on. Finding this kind of bugs is increasingly time consuming.

The increasing complexity of theory and programs is combined with the desire to analyse more complex behaviour and structures in more detail. So despite the exponential growth of computing power and resources, the demands for numerical analysis are out-pacing this development. In some numerical disciplines, like computational fluid dynamics, computational models too large and/or too complex for modern workstations are very common. In structural mechanics this tendency to analyse very large models is an upcoming trend, with the automotive industry (with e.g. car crash analysis) leading the way. This problem of using large models comes back to the code developer in the form of demands for ever more efficient and faster solution methods.

1.2 Research objectives

The objective of the work described in this thesis is to develop means to solve the problems mentioned above in computational structural mechanics: Improving the maintainability of the code combined with reduced

development time, and reducing the time of the analysis.

The development of new program structures to reduce the time in maintenance and development in computational mechanics is a recent field of research. The difference of the work described in this thesis with respect to others is the decision to use an *existing* finite element environment, and the idea that the main computational part of this code must be conserved and integrated in the new program structure. And, importantly, the resulting program should not perform slower than the already existing one.

A very effective way of reducing the total time of the analysis and of obtaining access to large resources, like memory, is by making use of parallel computers. This requires the design of a parallel version of the finite element code. In order to preserve the ease of development resulting from the new setup, this parallelism should be fully transparently implemented.

Finally, to demonstrate that the research is useful not only for academic problems, but also for industrial analysis, the program is used for existing and/or new applications. These applications reflect the use of finite element methods in university research and future industrial computations.

1.3 The finite element program

The developments of this research should be implemented in an existing finite element code. For that, use is made of an *explicit* finite element code, originally developed at the Delft University of Technology, the Netherlands. The code was chosen rather then an implicit code for several reasons. Firstly, in an explicit code the data and computations are tightly coupled, creating a large tendency to the creation of very complex code, which is difficult to develop and maintain. This makes an explicit code a suitable target for an improved code structure. Secondly, explicit codes are almost exclusively used for the simulation of dynamic, non-linear behaviour with complex models. Although this is a limitation to

the type of problems that can be solved, these kind of problems require extensive computational resources, thus making them among the first to benefit from parallel computing.

1.4 Layout of the thesis

This report has been organised as follows: First the theory of the finite element method is briefly discussed in chapter 2. The theory is limited to the explicit implementation and deals with some of the special problems of this method. Chapter 3 describes the principles of object oriented programming and discusses the way they can be used for finite element analysis with reference to existing literature on the subject. Chapter 4 then describes the object-oriented approach and the implementation in the developed code over the last three years. As the code has been designed to preserve the existing computational Fortran routines, it is 'wrapped' around the old code. Hence the name of 'object-oriented wrapper'. Chapters 5 and 6 deal with the implementation of new elements and material models in the code, as well as the theory behind the implementation of new developments.

Chapters 7 and 8 deal with the parallelisation of the explicit finite element code. Chapter 7 discusses some of the aspects of parallel computing, including hardware, as the software solutions are related to the targeted hardware. Chapter 8 describes the specific implementation of parallelism by mesh decomposition. An analysis is made of the potential performance, followed by a study of the behaviour of the program and of a comparison between theoretical and actual performance. Chapter 9 then shows some applications of the explicit finite element program. Section 9.1 and 9.3 deal with new territory for explicit methods. Section 9.2 shows a more common application, but applied to modern composite materials, using the material model described in 6.1. This is an application which needs parallel processing in order to keep computation times within acceptable levels.

2

Brief overview of explicit finite element theory

No development of finite element methods can be done without the knowledge of the theory of the method. Without this theoretical background, one has not a full understanding of the approximations made and the results produced.

This chapter gives a short overview of the theory of explicit finite element methods and their implementation. It tries to summarise the advantages and problems of the method. Starting with a brief description of general finite element theory, it will then describe some of the advantages and problems of the explicit implementation. A more extensive description of the explicit finite element method and its implementation can be found in [4, 9, 10].

2.1 The finite element method

In this section a brief overview of the theory of the finite element method is given. Using the principle of virtual work, the equations are derived from a general continuum in equilibrium. Using these equations, the finite element discretisation procedure is applied to obtain a finite set of linear equations. As the dynamic simulations often result in large displacements and rotations, subsection 2.1.3 describes the mathematics of frame-indifference. Finally, subsection 2.1.4 discusses some explicit time integration methods and stability requirements.

2.1.1 Equilibrium of a continuum

Let us consider a general body Ω in space in a time interval $[0, T]$ with boundary Γ and unit outward normal $\mathbf{n}$, as shown in figure 2.1. The

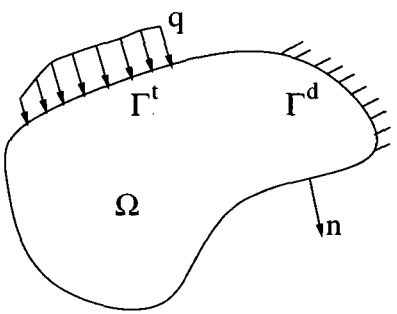


Figure 2.1: Deformable body in space

boundary Γ of the body Ω can be divided into two distinct parts:

$$\Gamma = \Gamma^d \cup \Gamma^t \tag{2.1}$$

where Γ^d is that part of the boundary where the displacements are prescribed and Γ^t that part of the boundary where the surface traction is

prescribed. We will omit the possibility of contact[1] for simplicity. To obtain the equations of equilibrium we can use the principle of stationary potential energy or the principle of virtual work. Here we will use the latter. The principle of virtual work states that:

> The total work of the external forces, including the inertial force, on any admissible virtual displacement field $\delta\mathbf{u}$ equals the total virtual work of the internal stress field on the virtual strain field corresponding to that virtual displacement field.

This principle can be written as:

$$\delta W_S = \delta W_E = \delta W_F + \delta W_I \tag{2.2}$$

where δW_S the virtual work of the internal stress field and δW_E the virtual work of the external forces, which can be split up into δW_F, the virtual work due to the forces acting on the body and δW_I the virtual work due to inertia.

The virtual work of a stress field $\boldsymbol{\sigma}$ on a virtual strain field $\delta\boldsymbol{\epsilon}$ is given by:

$$\delta W_S = \int_\Omega \boldsymbol{\sigma} \cdot \delta\boldsymbol{\epsilon}\, d\Omega \tag{2.3}$$

In case of elasticity the stress field is a function of the corresponding strain. In case of plasticity, creep or damage, the stress field can even be a function of the strain and the strain rate. The strain itself is again a function of the displacement. Therefore we should write in principle for the stress $\boldsymbol{\sigma}$:

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}[\boldsymbol{\epsilon}(\mathbf{u})] \tag{2.4}$$

Or, in case of material non-linearity:

$$\begin{aligned} \boldsymbol{\sigma} &= \boldsymbol{\sigma}(\boldsymbol{\epsilon}, \dot{\boldsymbol{\epsilon}}) \\ \boldsymbol{\epsilon} &= \boldsymbol{\epsilon}(\mathbf{u}) \end{aligned} \tag{2.5}$$

[1] For a description including contact refer to [19, 38, 42].

However, with this in mind we will use the short notation for brevity.

The virtual work of the forces acting on the body consists of forces acting on the volume and traction acting on the surface of the body:

$$\delta W_F = \int_\Omega \mathbf{b} \cdot \delta\mathbf{u}\, d\Omega + \int_{\Gamma^t} \mathbf{q} \cdot \delta\mathbf{u}\, d\Gamma \tag{2.6}$$

where $\mathbf{b}$ is the vector of the volumetric forces and $\mathbf{q}$ the surface traction. Finally, the virtual work due to inertia can be written as:

$$\delta W_I = -\int_\Omega \rho\ddot{\mathbf{u}} \cdot \delta\mathbf{u}\, d\Omega \tag{2.7}$$

where ρ is the density and $\ddot{\mathbf{u}}$ denotes the acceleration of the displacement field. Substituting these equations into Eq.(2.2) now gives us an expression for the equilibrium of a continuum:

$$\begin{aligned}\int_\Omega \boldsymbol{\sigma} \cdot \delta\boldsymbol{\epsilon}\, d\Omega - \int_\Omega \mathbf{b} \cdot \delta\mathbf{u}\, d\Omega \\ - \int_{\Gamma^t} \mathbf{q} \cdot \delta\mathbf{u}\, d\Gamma + \int_\Omega \rho\,\ddot{\mathbf{u}} \cdot \delta\mathbf{u}\, d\Omega = 0\end{aligned} \tag{2.8}$$

This equilibrium must hold for each time t at the interval $[0, T]$.

2.1.2 Finite element discretisation

Using the equilibrium of a continuous body we can obtain the equations of the finite element method. By dividing the body into small elements for which a solution or approximative solution can be found, we can compute the state of the entire body. Therefore, we assume that the displacement of an element can be approximated by the displacements of the nodal points:

$$\mathbf{u}(\mathbf{x}, t) = \phi_I(\mathbf{x})\mathbf{u}_I(t) \tag{2.9}$$

where $I = 1, 2..N$, N is the number of nodal points of the element, $\mathbf{U}_I$ the nodal displacements and ϕ_I the shape functions. This gives us for the velocity and acceleration of the element:

$$\dot{\mathbf{u}}(\mathbf{x}, t) = \phi_I(\mathbf{x})\dot{\mathbf{u}}_I(t) \tag{2.10}$$

$$\ddot{\mathbf{u}}(\mathbf{x}, t) = \phi_I(\mathbf{x})\ddot{\mathbf{u}}_I(t) \tag{2.11}$$

and, consequently:

$$\delta\mathbf{u} = \phi_I \delta\mathbf{u}_I \tag{2.12}$$

This can be written in matrix form as:

$$\mathbf{u}(\mathbf{x}, t) = a\,\mathbf{u}_e \tag{2.13}$$

where a is the matrix of the shape functions and $\mathbf{u}_e$ the vector of element nodal displacements.

With the displacements known at a given time t, the strains can be determined. Now assume that this relationship can be cast in a matrix form, such that

$$\boldsymbol{\epsilon} = S\mathbf{u} \tag{2.14}$$

where S is a linear operator. The strains can therefore be approximated by:

$$\boldsymbol{\epsilon} = b\mathbf{u}_e \tag{2.15}$$

where

$$b = S\,a \tag{2.16}$$

This approximation can be applied to the equation virtual work, so that the equilibrium for each element can be written as follows:

$$\begin{aligned}\int_{\Omega_e} b^T\boldsymbol{\sigma}\delta\mathbf{u}_e\,d\Omega - \int_{\Omega_e} a^T\mathbf{b}\delta\mathbf{u}_e\,d\Omega \\ - \int_{\Gamma_e^t} a^T\mathbf{q}\delta\mathbf{u}_e\,d\Gamma + \int_{\Omega_e} \rho a^T a\ddot{\mathbf{u}}_e\delta\mathbf{u}_e\,d\Omega = 0\end{aligned} \tag{2.17}$$

Which is valid for each allowed virtual displacement $\delta \mathbf{u}_e$. With this discretisation procedure, the equilibrium for each element at each time t at the interval $[0, T]$ can be written in matrix form:

$$m\ddot{\mathbf{u}}_e + \mathbf{r} = \mathbf{f} \tag{2.18}$$

where

$$m = \int_{\Omega_e} \rho a^T a \, d\Omega \tag{2.19}$$

$$\mathbf{f} = \int_{\Omega_e} a^T \mathbf{b} \, d\Omega + \int_{\Gamma^t} a^T \mathbf{q} \, d\Gamma \tag{2.20}$$

and

$$\mathbf{r} = \int b^T \boldsymbol{\sigma} \, d\Omega \tag{2.21}$$

Assembly of all element vectors and matrices gives the global equilibrium equation in discretised form:

$$M\ddot{\mathbf{U}} + \mathbf{R} = \mathbf{F} \tag{2.22}$$

where $M = \sum m$, $\mathbf{U} = \sum \mathbf{u}_e$, etc. and $\sum$ denotes standard finite element assembly. The vector of internal forces is of course in general a function of the displacements and velocities as shown in Eq.(2.4) or Eq.(2.5), so that we should write $\mathbf{R} = \mathbf{R}(\mathbf{U}, \dot{\mathbf{U}})$.

Linearisation

As an illustration on how to solve the more general equations above, and to arrive to more familiar equations, we consider only linear elasticity and small rotations. In case of small displacements and small strains, we can assume that we can write the constitutive relation as:

$$\boldsymbol{\sigma} = C\boldsymbol{\epsilon} = Cb\mathbf{u}_e \tag{2.23}$$

so that for the internal forces, Eq.(2.21), we can write:

$$\mathbf{r} = k\mathbf{u}_e = \int_{\Omega_e} b^T C b \, d\Omega \cdot \mathbf{u}_e \tag{2.24}$$

where k the element stiffness matrix. Assembly of the global system gives the more familiar expression (usually for implicit codes):

$$M\ddot{\mathbf{U}} + K\mathbf{U} = \mathbf{F} \tag{2.25}$$

2.1.3 Large displacements and rotations

The resulting discretised equations (2.25) describe the *dynamic* equilibrium of the continuum, which includes the possibility of large displacements and rotations. This means that the computed stresses must be independent of any superimposed rigid body motion or change of reference frame. This is called *frame-indifference* or *objectivity*. For any point with coordinates $\mathbf{x}$ the new coordinates $\tilde{\mathbf{x}}$ after a superimposed rigid body motion are given by:

$$\tilde{\mathbf{x}} = \mathbf{c}(t) + Q(t)\mathbf{x} \tag{2.26}$$

where $\mathbf{c}(t)$ is a translation and $Q(t)$ an orthogonal rotation tensor:

$$Q^{-1} = Q^T \quad \text{and} \quad Q \cdot Q^T = Q^T \cdot Q = 1 \tag{2.27}$$

or in index notation:

$$Q_{ik}Q_{jk} = Q_{ki}Q_{kj} = \delta_{kj} \tag{2.28}$$

We now say that a vector $\mathbf{v}$ is objective when

$$\tilde{\mathbf{v}} = Q(t)\mathbf{v} \tag{2.29}$$

and a second order tensor T when

$$\tilde{T} = Q(t)T\,Q(t) \tag{2.30}$$

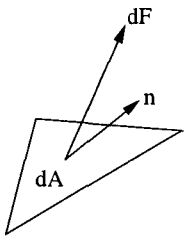


Figure 2.2: Definition of the Cauchy stress

More generally, a tensor is objective when, using index notation:

$$\tilde{T}_{ij\ldots k} = Q_{il}Q_{jm}\ldots Q_{kn}T_{lm\ldots n} \tag{2.31}$$

We shall now show that the stress tensor generally used in mechanical engineering, the Cauchy stress tensor, or engineering stress, is an objective tensor.

The Cauchy stress is defined by:

$$d\mathbf{F} = \boldsymbol{\sigma}\, dA\, \mathbf{n} \tag{2.32}$$

where $d\mathbf{F}$ is the force vector, dA the surface and $\mathbf{n}$ the unit outward normal, as shown in figure (2.2). This can be written in index notation as:

$$dF_i = \sigma_{ij} dA_j \tag{2.33}$$

The force and the surface are rotated into their new coordinates $d\tilde{F}_i$ and $d\tilde{A}_i$ as follows:

$$d\tilde{F}_i = Q_{ij} dF_j \tag{2.34}$$

$$d\tilde{A}_i = Q_{ij} dA_j \tag{2.35}$$

The components of the stress tensor in the transformed system are defined by

$$d\tilde{F}_i = \tilde{\sigma}_{ij} d\tilde{A}_j \tag{2.36}$$

Combining these equations yields:

$$\begin{aligned} d\tilde{F}_i &= Q_{ij} dF_j = Q_{ij}\sigma_{jk} dA_k \\ &= \tilde{\sigma}_{ij} d\tilde{A}_j = \tilde{\sigma}_{ij} Q_{jk} dA_k \end{aligned} \tag{2.37}$$

Since this must be valid for all dA_k we get:

$$\tilde{\sigma}_{ij} = Q_{ik}\sigma_{kl}Q_{jl} \tag{2.38}$$

or in matrix notation

$$\tilde{\sigma} = Q\sigma Q^T \tag{2.39}$$

which shows that the Cauchy stress is objective.

2.1.4 Time integration

Here above we have derived the general equations for the finite element method. The resulting matrix form of the equilibrium equation, Eq.(2.25), is the dynamic form, describing the behaviour of a continuum in space and time. Using the finite element method we obtain a large set of linear equations which allows us to solve the displacements, but for the total solution we need to integrate the equations in time. The time domain to be considered, $[0, T]$, is divided into subdomains for an incremental procedure, for which there is a vast choice of integration methods, but we will consider here only some of the so-called *explicit* integration schemes.

Newmark methods

Consider a subsequent set of points in time, denoted by t_{i-1}, t_i and t_{i+1}, respectively, and let the time step at t_i be denoted by $\Delta t_i = t_i - t_{i-1}$.

Then the Newmark methods relate the acceleration, velocity and displacement as follows:

$$m\,a_{i+1} + c\,v_{i+1} + k\,u_{i+1} = f_{i+1} \tag{2.40}$$

$$u_{i+1} = u_i + \Delta t_i v_i + \tfrac{1}{2}\Delta t_i^2[(1-2\beta)a_i + 2\beta a_{i+1}] \tag{2.41}$$

$$v_{i+1} = u_i + \Delta t_i[(1-\gamma)a_i + \gamma a_{i+1}] \tag{2.42}$$

where m is the mass, c the damping and k the stiffness factor or matrix. β and γ depend on the application and $u_i = u(t_i)$ etc for brevity. Notice that for $\beta \neq 0$ the solution of the equation at time t_i is required to obtain the displacement at time t_i. This classifies the method as an implicit method and is therefore of no interest here. For $\beta = 0$ and $\gamma = \frac{1}{2}$ the method reduces to the so-called explicit *finite difference* scheme:

$$u_{i+1} = u_i + \Delta t_i v_i + \tfrac{1}{2}\Delta t_i^2 a_i \tag{2.43}$$

$$v_{i+1} = v_i + \tfrac{1}{2}\Delta t_i[a_i + a_{i+1}] \tag{2.44}$$

For a small perturbation in the solution to remain small, the time step must remain under a critical value. The stability condition depends on the chosen integration method. The critical time step for the Newmark method is:

$$\Delta t < \infty \qquad \text{if} \quad 2\beta \geq \gamma \geq \tfrac{1}{2} \tag{2.45}$$

$$\Delta t \leq \frac{T_{min}}{2\pi\sqrt{\frac{\gamma}{2} - \beta}} \qquad \text{if} \quad \beta < \tfrac{\gamma}{2} \quad \text{and} \quad \gamma \geq \tfrac{1}{2} \tag{2.46}$$

where T_{min} is the smallest period in the system. Although some of the implicit methods can be unconditionally stable, the time step will be limited due to accuracy requirements and convergence problems due to non-linearities. For more on time integration schemes and their stability refer to [14].

Predictor-corrector methods

For non-linear analysis the explicit predictor-corrector methods are used. These are based on the following set of equations. First, a prediction for

the displacement, $\hat{u}_{i+1}$, and the velocity, $\hat{v}_{i+1}$, is made:

$$\hat{u}_{i+1} = u_i + \Delta t\, v_i + \tfrac{1}{2}\Delta t_i^2(1-2\beta)a_i \tag{2.47}$$

$$\hat{v}_{i+1} = v_i + \Delta t(1-\gamma)a_i \tag{2.48}$$

$$\tag{2.49}$$

after which the new acceleration is computed using the predicted displacement and velocity, according to:

$$m\, a_{i+1} + c\, \hat{v}_{i+1} + k\, \hat{u}_{i+1} = f_{i+1} \tag{2.50}$$

Now the displacement and velocity are corrected by:

$$u_{i+1} = \hat{u}_{i+1} + \tfrac{1}{2}\Delta t_i^2 2\beta a_{i+1} \tag{2.51}$$

$$v_{i+1} = \hat{v}_{i+1} + \Delta t_i \gamma a_{i+1} \tag{2.52}$$

Notice that Eqs.(2.47) and (2.48) are equal to Eqs.(2.41) and (2.42). The predictor-corrector methods now use the predicted values for the velocity and the displacement for the equation of motion (2.50). It can be shown [14] that the predictor-corrector methods are conditionally stable when

$$\gamma \geq \tfrac{1}{2} \tag{2.53}$$

$$\omega\, \Delta t \leq \frac{\sqrt{\xi^2 + 2\gamma} - \xi}{\gamma} \tag{2.54}$$

where ω^2 is the maximum eigenvalue in the system and ξ the modal damping ratio:

$$\xi = \tfrac{1}{2}\left(\tfrac{a}{\omega} + b\,\omega\right) \tag{2.55}$$

if

$$c = a\, m + b\, k \tag{2.56}$$

We can see that the critical time step decreases with ξ. Therefore, the minimum is found when $\xi = 0$:

$$\omega \Delta t \leq \sqrt{\frac{2}{\gamma}} \tag{2.57}$$

with a maximum $\Delta t = \frac{2}{\omega}$ for $\gamma = \frac{1}{2}$, which is the same result as for the central difference scheme. It can be shown that the predictor-corrector method and the corresponding Newmark method are identical when there is no damping present.

2.2 Explicit implementation

In the above section the derivation of the basic theory for explicit finite element analysis has been shown. In following section the aspects of the implementation into a computer algorithm are discussed. In subsection 2.2.1 the implementation and advantage of the explicit time integration are discussed. In subsection 2.2.2 one of the most typical features of the method is described, i.e. avoiding to compute the stiffness matrix, after which in subsection 2.2.2 the problems with numerical integration of elements are described. Finally, subsection 2.2.3 summarises briefly the complete scheme to be implemented in an explicit finite element program.

2.2.1 Equations of motion

The explicit finite element method uses the predictor-corrector method to integrate the equations in time, the latter being very well suited for non-linear analysis. The advantage of this particular method is that the explicit time integration methods are very simple and efficient. The drawback is the stability requirement, determining the maximum time step to be very small. Explicit methods are therefore exclusively used to simulate phenomena occurring in a small time interval. This means that for the

predictor-corrector method with $\beta = 0$ and $\gamma = \frac{1}{2}$, the following set of equations needs to be solved:

$$\ddot{\mathbf{U}}_i = M^{-1}(\mathbf{F}_i - K\mathbf{U}_i) \tag{2.58}$$

$$\dot{\mathbf{U}}_{i+1/2} = \dot{\mathbf{U}}_{i-1/2} + \Delta t_i \, \ddot{\mathbf{U}}_i \tag{2.58.b}$$

$$\mathbf{U}_{i+1} = \mathbf{U}_i + \Delta t_i \dot{\mathbf{U}}_{i+1/2} + \tfrac{1}{2}\Delta t_i^2 \ddot{\mathbf{U}}_i \tag{2.58.c}$$

where we have introduced the notation $\dot{u}_{i+1/2} = \hat{\dot{u}}_{i+1}$. Apart from Eq.(2.58) these are all sets of un-coupled equations. In order to decouple Eq.(2.58) we rewrite it as:

$$\ddot{\mathbf{U}}_i = M^{-1}(\mathbf{F}_i^{ext} - \mathbf{F}_i^{int}) \tag{2.59}$$

where $\mathbf{F}^{ext}$ is the external force vector and $\mathbf{F}^{int}$ the internal force vector or $K\mathbf{U}$. If the mass matrix is diagonal, the so-called lumped mass, then M^{-1} is diagonal and Eq.(2.59) is decoupled.

Recalling the stability requirement for the integration method, Eq.(2.57), with $\gamma = \frac{1}{2}$

$$\Delta t \leq \frac{2}{\omega} \tag{2.60}$$

where ω^2 is the largest eigenvalue of the system. The stability requirement demands the use of very small time steps for the time integration. This is an important limitation of the method. It requires the code to be implemented with computational speed per cycle as an almost dominating parameter. As it turns out, the most time-consuming part of the computation is the calculation of the internal forces and not the integration of the equations of motion.

The advantage of decoupling this set of equations is that the resulting algorithm is simple and very fast. It also allows for the integration procedure to be vectorised easily or parallelised in order to run on vector or parallel computers.

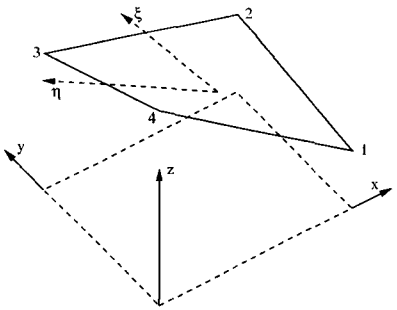


Figure 2.3: General flat element mapped into three dimensions

2.2.2 Element integration

The integral equation for the internal forces, Eq.(2.21), cannot generally be solved analytically, but has to be integrated numerically [2, 14, 43]. As shown in figure 2.3, the general shell in three dimensions is mapped onto a two dimensional surface by means of iso-parametric shape functions. Using Gauss integration the strains are computed in several reference points, the so-called Gauss points. For a 4-node shell element with bilinear shape functions one needs two by two integration points over the surface to properly represent the polynomial used to map the element. However, this bilinear approximation results in additional shear stiffness in case of pure bending of the membrane. This so-called "locking phenomena" is demonstrated in figure 2.4. The nodal displacements due to the pure bending of the element are the same as the nodal displacements which cause shear. However, according to beam theory, pure bending does not result in shear strain. But as the standard shell theory does not take the out-of-plane rotations into account, this distinction between the two modes can not be made, resulting in a too stiff response of the shell element in pure bending. In order to prevent this phenomena from affecting the solution, additional schemes need to be implemented.

Because computation time is an important factor in explicit finite el-

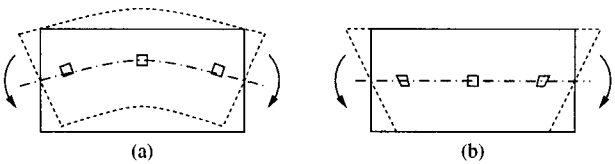


Figure 2.4: Membrane locking illustrated: (a): pure bending; (b): linear displacements causing shear.

ement analysis, one-point integration of the element is generally used. Although it reduces the accuracy of the solution, it not only reduces the amount of computations by computing the strains in only one point instead of four, but also eliminates the above mentioned locking problem. However, it turns out that by using this simplified method, the element can deform in a specific shape without restrictions. This is called a zero-energy or hourglass mode of the element and has be avoided.

Hourglass modes

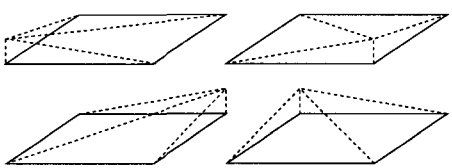


Figure 2.5: Isoparametric shape functions of a four node shell element

The shape functions ϕ_N of isoparametric elements are given by Lagrange polynomials. For the four node quadrilateral we get:

$$\phi_N = \tfrac{1}{4}(1 \pm \xi)(1 \pm \eta) \tag{2.61}$$

These shape functions are shown in figure 2.5. However, due to the under-integration of the element, deformations without resulting stresses

can occur resulting in large deformations of the element and destroying the solution. These modes are called *hourglass* or *zero-energy* modes. To understand these modes, we have to look differently at the shape functions. It can be shown that these modes originate from the absence of bilinear terms in the deformation or velocity field. According to Belytschko [8], the shape functions can be written as an orthogonal set of base vectors:

$$\phi_N = \tfrac{1}{4}(\Sigma_N + \xi\Lambda^1_N + \eta\Lambda^2_N - \xi\eta\Gamma_N) \tag{2.62}$$

where Σ_N can be identified as a rigid body translation, Λ^i_N the linear normal strain modes and Γ_N the hourglass mode. The values for them are:

N	Σ	Λ^1	Λ^2	Γ
1	1	-1	-1	-1
2	1	1	-1	1
3	1	1	1	-1
4	1	-1	1	1

These modes are shown in figure 2.6. It turns out that when using one-point quadrature to integrate the virtual work of the internal forces, the terms including the shape functions Σ and Λ give a contribution to the integral of zero.

In order to prevent these deformations from growing without contributing to the internal energy, both additional damping and additional

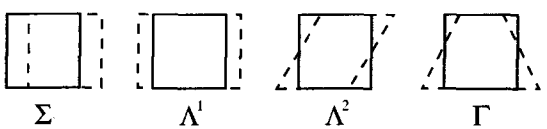


Figure 2.6: Orthogonal shape functions of an isoparametric shell element

stiffness schemes have been proposed, separately as well as in combination. The problem with introducing additional damping is that this mode is still allowed to grow, resulting in permanent deformation of the element. Introducing additional stiffness to allow only mild hourglassing is now considered to be the most effective solution. This technique results in a limited amount of additional coding, while giving good results. It is therefore used in almost all modern codes.

A more detailed description of the implementation of an element with one-point integration and hourglass control can be found in [8] and in section 5.1, which deals with the implementation of an eight-node volume element with hourglass control.

2.2.3 Implementation scheme

The theory and algorithms described above can now be implemented in a computer program. The resulting scheme of the simulation now turns out as shown in figure 2.7.

In a loop over all the elements the internal forces are computed. Then, in two loops over the nodes the equations of motions are integrated. Based on the smallest critical time step in the system the time is updated. If the end time of the simulations has been reached, the integration stops, otherwise it returns to the beginning for the next cycle.

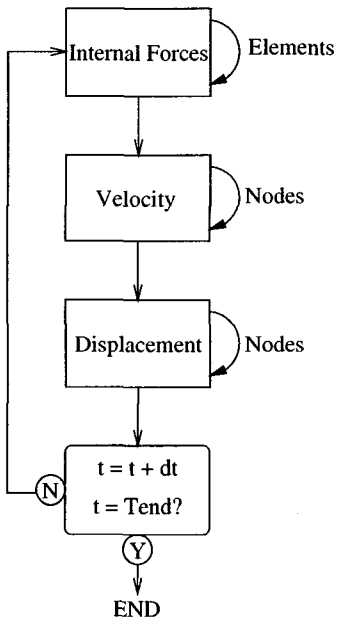


Figure 2.7: Implementation scheme of an explicit finite element program

3

The object oriented approach

This chapter describes our considerations with respect to the object oriented approach and the structures chosen for the new program modules. First, the principles of object-oriented programming are described briefly in section 3.1. After a literature study to the use of object-oriented programming for finite element program, section 3.2, section 3.3 discusses the reasons for and the way of applying the object-oriented approach. Next, in section 3.4 the finite element program B2000 is introduced.

3.1 Object-oriented programming

In this section the principles of object-oriented programming (OOP) are briefly discussed. It is based on chapter 11 of [32]. For more information on OOP see [32], or the book of Bjarne Stroustrup [36].

Since the introduction of the digital computer in the 1940s, programming and programming languages have come a long way. First, program-

ming was done directly by binary instructions. With the increasing complexity of the programs came the development of the assembly language (so-called low-level programming), and later the high-level languages, of which Fortran was and still is the most well known and widespread in the scientific and engineering community. The 1960s saw the development of the middle-level languages, like the C language, which combined the best of both worlds. However, today, with the increasing complexity of the programs (with over 100,000 lines of code), even the structured programming approach offered by C is no longer sufficient to maintain an overview of the entire program. In order to solve these problems a new programming paradigm, called object-oriented programming has been devised.

Object-oriented programming requires a whole new way of thinking about programming, combining the best ideas from structured programming and adding new concepts to deal with greater complexity. Instead of developing the algorithm first and starting to program directly, the developer should first think about the structure of the program. The flow diagram of the two processes is shown in Figure 3.1. The advantage of object-oriented programming is that, once the program structure is set up properly, new code can find its place immediately. The programmer has to adopt a different way of thinking about his program, dividing the tasks at hand into different subtasks that take into account both the data and the code related to each group. The groups have to be organised in a hierarchical structure and put into self-contained parts, the so-called objects. All the object-oriented programming languages offer three new concepts: encapsulation, polymorphism and inheritance, which will be considered here briefly.

3.1.1 Encapsulation

The mechanism of binding together code and data and of protecting it from the outside is called *encapsulation*. This is the way to create 'objects' (hence object-oriented programming), entities which contain data

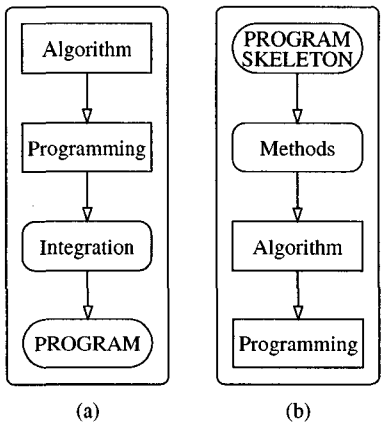


Figure 3.1: Design flow of sequential (a) and object-oriented (b) programming

as well as code to manipulate the data. The data may be made inaccessible from the outside in order to prevent other parts of the program of accidentally modifying it.

3.1.2 Polymorphism

The mechanism of 'one interface, multiple methods' is called *polymorphism*. It is used to allow one interface (e.g. one function call) to be used for a general class of actions. This helps to reduce the complexity of the program. For example, a sorting algorithm might be different for integers and floating points. Using polymorphism the two different functions can both be called `sort()`.

3.1.3 Inheritance

Inheritance is the principle of using the properties and/or functions of another object for a new object. Thus objects can be *classified* and a certain hierarchy can be created. For example (from [32], chapter 11) "a Red Delicious apple is part of the classification *apple*, which in turn is part of the *fruit* class, which is under the larger class *food*." As will be shown, inheritance is an important and useful feature of object-oriented programming.

3.1.4 Code reuse

One of the other terms associated with object-oriented programming is *code reuse*. The obvious meaning of this would be the possibility to reuse code for other parts or programs, once it has been written. Fortran is known for being a programming language people use to write programs which are used only once (to write a set of input files for a test case, for example) and then thrown away. Writing proper object-oriented code is never done quickly and requires a lot of thought, so that a good designed piece of code will get reused for other versions or similar programs. This gives us the real meaning of code reuse: namely the reuse of the *structure* or *setup* of the code, not necessarily the lines of the algorithm themselves. Hence, the success of an object-oriented design can be partially measured in terms of reuse of the ideas incorporated in the design for other, similar programs.

3.2 Literature study - current state

At the beginning of this thesis a study of current literature on object-oriented programming for finite element methods was carried out. As this field is quite new and lies on the border of computer science and computational mechanics, very little material was found. The articles on

this topic have one thing in common: They consider a complete re-design of finite element codes according to the object-oriented principles.

Making use of the object-oriented programming techniques as described above, the resulting programs as found in the literature are very fine grained in order to make implementation of new features, like elements, as easy and quick as possible. As an example, the following inheritance structure is created for a single element [6, 45], as shown in Figure 3.2:

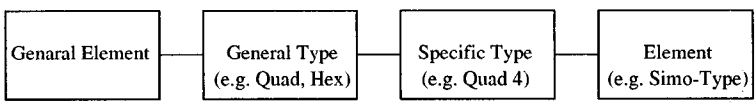


Figure 3.2: Example of an inheritance structure for FEM: element

The implementation of a new element now consists just of introducing the last part, `element`, which contains exclusively only the specific tasks for this particular element type. The advantage of this approach is that for the implementation of a new element coding is reduced to a minimum as most parts are already in place.

Such a finite element program approach can be said to be really object-oriented, but it has, however, three main drawbacks. The first is that the concept itself is completely incompatible with the current coding techniques and existing codes. Therefore, a complete rewrite of the finite element program is required. The second drawback is the overhead generated by this scheme, as each element is considered to be unique. Each function call generates overhead and clears the cache-line of the CPU, thereby causing the execution time to be much slower than with the existing (Fortran 77) codes. No use is made of the logical structure in a finite element model, like having more than one element of the same type and grouping them together for efficiency. The third disadvantage is related to the previous one: If an element-object is created for each element, much care must be taken in the implementation of allocation of memory. As a carelessly applied `new` operator call takes approximately

80 ms, a new for each element would take too much time. For a problem with 100,000 elements, this would result in 130 minutes just for the allocation of the elements! Of course proper programming can avoid this, but the idea should be clear: Dynamic implementations of objects can create significantly slower code. Reorganising the elements, and collecting elements of the same type in so-called containers or otherwise, would create a much more efficient program. An example of such an inefficient implementation is given in [5], where for larger problems with up to 2700 elements the increase of CPU time compared to the original Fortran code is up to 60%! Although the author considers this acceptable because of the time gained in program development, such an increase in CPU time would *not* be acceptable for any industrial application.

The goal of this part of the thesis is therefore to design an object-oriented or object-oriented-like code which eliminates these drawbacks. So before designing a new concept for finite element methods, let us first consider the demands on and constraints for such a concept.

3.3 Considerations for the setup

Before we can start designing new program structures for finite element programs, we have to take some facts into account. We must consider the setup and contents of existing programs and programming paradigms, data structures and, not unimportant, the profile of the programmers themselves. Based on these considerations we have to decide on a programming language which will suit our needs, as the programming language itself will determine to a certain extent our programming possibilities. Then, with the existing structures and the new language, we can start to design our new setup.

3.3.1 The existing environment

The existing programming environment is closely linked to the existing programming language, usually Fortran. Although some programs or

parts of programs are written in C, Fortran is still by far the most widely used programming language in scientific computing. This leads us to the following considerations:

- Computational routines are written in Fortran.
- Data is organised both in `common` blocks and sometimes in simple dynamic memory structures.
- Programmers are used to program this way
- Resulting executable may not be much slower then the existing one.

The fact that most (or all) computational routines are written in Fortran means that we must keep these Fortran subroutines and find a way to integrate them into the new code, because it would be too much work to rewrite the code for 50 to 100 (or even more) different element types and several different solvers. One single element routine may represent many years of work. The unstructured code on top of these routines must be (re)structured. The rearrangement of data will probably cause the greatest conflict with the object-oriented programming paradigm.

The fact that the current programmers of finite element codes are used to program in the serial, unstructured way of Fortran 77 must not be overlooked, as they will have to use the new code and continue to develop it. This means that the programmers must be able to continue programming (for example introduce new element types) in Fortran to make the transition from procedural programming to object-oriented programming more gradual.

The possible increase in CPU time should be kept within limits in order to gain acceptance for the new method. Object-oriented programming creates in general more overhead in the code than the procedural programming style, but we must bear in mind that a considerable increase of execution time would not be accepted by the users.

3.3.2 The programming language

In order to write an object-oriented program we need a programming language which supports the mechanisms described in Section 3.1. The three possible options to be considered are:

- Java
- Fortran 90 (or its successor)
- C++

Although Java now has a C interface, the language is still under development and the resulting executable code too slow (an order of magnitude) to be a viable option at the moment. Because it is too much of a pure object-oriented programming language with totally different paradigms, the integration of the existing Fortran subroutines is too complicated. Also, getting Java accepted by the programmers of numerical codes might be problematic due to its development status and the totally different syntax and style. However, the use of Java might be interesting for future research. New compilers creating machine dependent executables show promising results in execution times, but these are still limited and experimental.

The Fortran 90 language undoubtedly has the best interface for the Fortran 77 code and changing from Fortran 77 to Fortran 90 is relatively easy for current programmers. However, the compilers are still under development creating slow executables as they are not considered high priority by compiler developers. Also, the language is not really object-oriented, but merely a further development (some would say 'patched version') of the old Fortran 77.

The only viable option which then remains is C++, which is an object-oriented language based on the C language with a very similar syntax. Although far from perfect[1], it is a real object-oriented language, it has a

[1] With C++, consider the following remark from the founder of C++, Bjarne Stroustrup: "C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg."

good interface for Fortran 77 (and of course C) and produces fast executables. Thanks to its C-like syntax it is less alien for current Fortran programmers, although changing from one language to the other takes a real effort. As C++ allows for sequential programming it does represent a risk of people simply changing their programming syntax, but not their programming style. In order to avoid this, the proposed new code must be set up in such a rigid way that procedural code-writing is practically impossible.

3.3.3 The building blocks

With the overview of the current program situation and with C++ as the new programming language for object-oriented programming we have to define the building blocks of our finite element method. For that we will briefly consider the modelling of the structure of the computational model.

If we define a computational model, we distinguish between the geometry and its associated mesh, the boundary conditions, and the external forces (right-hand sides). The mesh is usually divided into subdomains, or branches, which are divided into cells (elements), like shells and beams. The operators on these cells are then again divided into separate elements of the same type. The smallest unit in a finite element computation is therefore the element[2]. However, making an element the smallest object in the object-oriented code would cause conflicts with the existing data structures, where data like node connectivity lists and nodal coordinate lists are stored in long global arrays. Implementing this in a truly object-oriented way would create highly inefficient code due to cache misses and overhead. The next level to be considered is then the element type. Defining the element type as the smallest unit would be appropriate as, strictly speaking, one does not develop a new element,

[2]Strictly speaking, the smallest unit would be the node. However, the idea of creating an object for each node implies the creation of millions of objects in case of large models, basically killing every performance of the computer due to the demand of resources. Also, this is completely incompatible with the existing data structures.

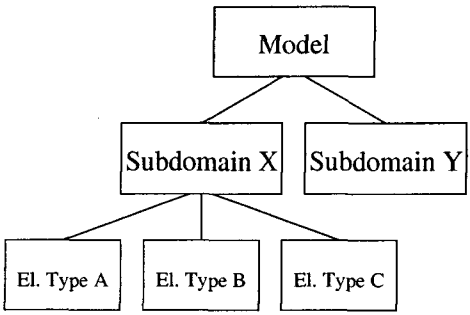


Figure 3.3: Basic hierarchy of a FE code

but a new element type. Also, part of the data structure is already organised by element type, like node connectivity lists, material properties, etc. Furthermore, it is relatively easy to reorganise data, like material number lists in such a way that it is no longer a global array, but instead element type based.

So an element type object would contain the data and the code of all elements of its type. A subdomain is then made out of a collection of element types and additional information, like boundary conditions and external forces. The model itself is made of several different subdomains. The resulting hierarchy is depicted in Figure 3.3. With this global idea in mind, we can start developing a new layout for the different modules of the finite element code.

3.4 Introduction to B2000

Here the B2000 finite element program which we have used to implement the developed methods, is introduced. The first part of this section

is taken from [39]. For more information about B2000[3] one is referred to [22, 23, 39].

B2000 is a modular program system for solving discrete problems such as the finite element method. Although the code could be used to solve any type of discrete problem, we will here only consider the finite element part, as this is the method of interest. Because of its modular architecture, programming in B2000 is relatively simple, providing a high visibility and easy access to the information required for proper analysis. Thus, B2000 is ideally suited for a research environment.

The modular nature of B2000 breaks down the computations which have to be performed in a finite element analysis into its elementary tasks, such as the formulation of the element stiffness matrix, the assembly of the global matrices and the solution of the equations. Every step is performed by its own processor (module). Computations can now be done step-by-step using one module at the time, or in execution of a macro-processor which integrates several processors in one program. The first method is useful for certain kinds of research as well as debugging, while the latter is preferred for production purposes.

The foundation of B2000 is the database management system MemCom which controls the data, stores data in a logical database, allocates memory etc. Therefore, the programmer can focus more on the real solution of the problem itself and less on the annoying part of data management. The basis of the program was originally developed at SMR, based on the modular programming model coming from Peter Stehlin and Silvio Merazzi [21]. This program soon found its way to universities and research institutes. Now, the resulting program B2000 is used and developed by most European aerospace laboratories and several universities.

B2000 is written in Fortran 77, so this will enable us to investigate the integration of the old code with the new programming meth-

[3] B2000 papers and documentation published by SMR can be found on the SMR homepage: http://www.smr.ch

ods. The reason for B2000 being very well suited for the development of object-oriented programming in finite element programs is its modular setup. Because of that, we can take only one of the modules to develop a prototype, instead of having to deal with all the complexities of a finite element code at once. One of the modules is the recently added explicit time integrator B2ETA (Explicit Transient Analysis processor, [9, 10, 38, 40, 41]), which can be considered a more or less independent program. Since the development of this program is very recent and the code mainly written in C, the program is relatively small and therefore very well suited for rewriting. Thus we are able to investigate the possibilities of object-oriented programming in finite element codes without the additional complication of the Fortran integration, as C code is much easier to integrate into C++ (see page 30).

4

The design of the object-oriented wrapper

The main problem with existing finite element programs is the maintenance of the code. This is due to the programming languages used: Fortran and C are procedural programming languages, which implies an increasing difficulty maintaining code as size and complexity grow. This means that there are increasing risks of introducing serious errors, and that new developments take more time to implement. In order to overcome these problems, object-oriented programming has been invented. This allows data to be *encapsulated* in objects to prevent accidental access and modification. It also makes code much more readable thanks to the data being locally in the object.

However, object-oriented programming cannot directly be applied to existing finite element codes, as it is incompatible with the sequential programming method. But the existing code cannot be thrown away, since too much time and effort has been spent on developing it. Therefore we have designed the *object-oriented wrapper* for finite element codes and applied it to an explicit finite element code written in C. Due to the good

interface with C and Fortran, C++ was chosen as the new programming language.

The design and implementation of the so-called object-oriented wrapper is the central part of this thesis. The term 'wrapper' stands for the idea of a shell created around the existing program structure in order to keep and protect existing routines by encapsulating them in an object-oriented finite element program. The principle of integrating the existing Fortran routines in the new object-oriented code is the main difference between the work presented here and other object-oriented finite element implementations found in literature. The reason for opting for an object-oriented program is ease of maintainability and development. However, contrary to the opinions expressed in most of the articles found on this subject, we are here brought to the conclusion that the substantial loss of performance of the new program structure created this way, is not acceptable.

This chapter describes the design of the object-oriented wrapper for combining object-oriented code and sequential code and its implementation in the explicit transient analysis processor of B2000, B2ETA . The ideas behind the new structure are explained and a scheme of the new setup given. Finally, some initial results are presented.

4.1 FE Model and Explicit Method

On the basis of the design of the object-oriented structure is the model for the finite element method for structural mechanics. A structure is discretised into small elements of different types (see figure 4.1). The cover consists of some bars to support the plates. This is the physical idea behind the finite element method.

In reality, the models are more complex than the one described above. Large complex models are made of different parts (subdomains) which each are made of elements and nodes, as shown in figure 4.2: The panel and the stiffener are two different domains with their own mesh of (four-node) elements.

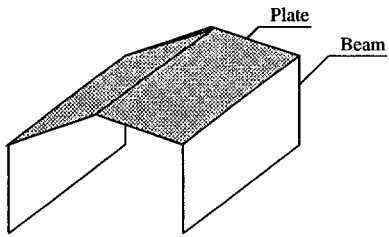


Figure 4.1: A structure made up of different elements

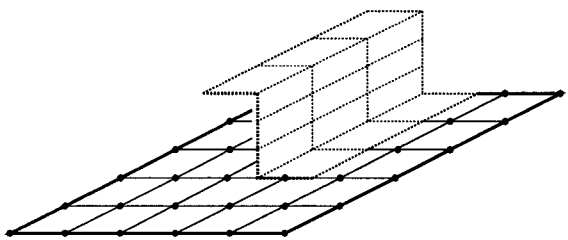


Figure 4.2: A model with subdomains, elements and nodes

Looking at this physical explication of the finite element method we can see that there are two different parts in a model: Nodes and elements. Everything else is node or element based: Boundary conditions, forces, pressure, etc. Therefore, the smallest and most general objects would be nodes and elements. However, from the point of efficiency, this is not considered practical. Nodes themselves are numerous and very small. By grouping them together into one object we obtain much more efficient code. Elements are developed as an element type. Creating an object "element type" as the smallest element object then seems more logical and also allows for easier integration of the existing Fortran code with the object-oriented wrapper. By putting all elements of the same type in one object we once again create more efficient code. Grouping all

elements of the same type in one object allows for much more flexibility as will be shown in the case of the implementation of discrete elements in section 5.2.

4.2 The object-oriented wrapper

Considering the complexities of combining object-oriented and sequential code, the concept of 'wrapping' the existing code in an object-oriented program framework has been developed, where the wrapper is like a shell around the procedural code. The setup of the object-oriented program is based on the natural division within a finite element model: Nodes, elements and properties. The actual implementation provides the *interface* of the methods to the core of the program while leaving its actual implementation free and hidden. The implementation of methods can be done by calling the existing (sequential) Fortran routines, by integrating C functions of by any other (object-oriented) implementation.

This design requires a complete rewrite of the *core* of the finite element program and a solid design of the classes and their interface. However, the bulk of a finite element code is in the programming of the methods (element types, solvers, etc.) which can be kept in their original form, except for the fact that all data previously stored in `common`-blocks need to be passed in a different way. The most direct method is by passing the used variables as arguments of the function.

The resulting scheme of the complete general finite element program is shown in figure 4.3. The number at the beginning of an arrow denotes the number of references to the underlying objects. The notation $n+$ denotes n or more references. The main module controls the computation, creates the required number of subdomains and loads the material data. However, the controlling module does not need this data, hence the dotted line. Material data is only loaded by the module in order to prevent duplication of data. Each subdomain creates the nodes and element types. The elements have a reference to the material object in order to obtain the material data. Each element type class is created from the

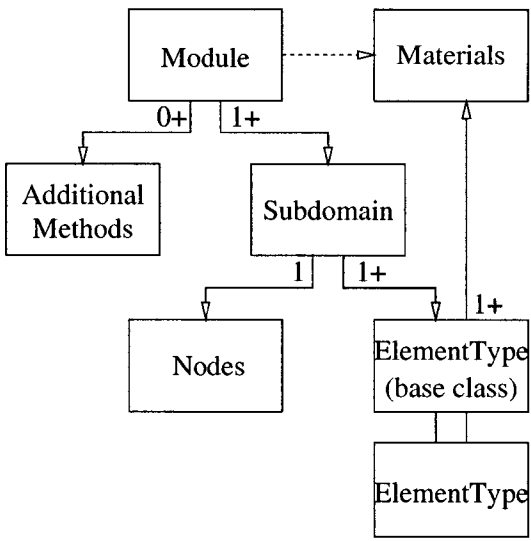


Figure 4.3: Scheme of an object-oriented FE code

same elementary base class.

Figure 4.4 shows the idea of the wrapper for the element type methods in more detail. The element type base class may contain some general data, but its main purpose is to define the *interface* of all element types for the rest of the program (e.g. `computeStiffnessMatrix()`, `computeMass()`). This is implemented in C++ by using *virtual functions*. This is an advanced form of polymorphism (see 3.1.2): The function of the base class points to the actual implementation of this function in the derived class with the same function name.

For each element type a derived class (inheritance, see 3.1.3) is created which contains only the data required for the element type, and which implements the functionality for all elements of this type. The specific implementation is not prescribed. This can be done using the

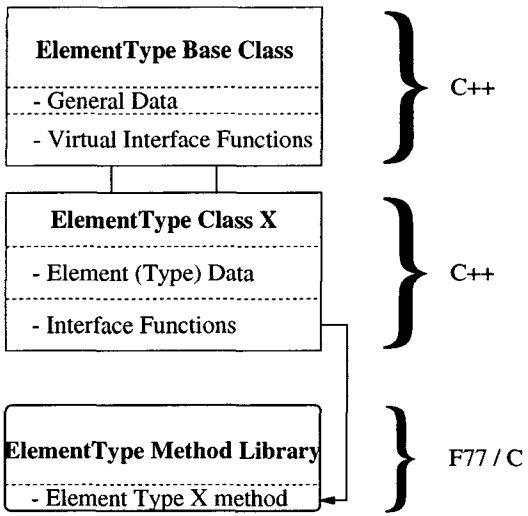


Figure 4.4: Fortran wrapper of element methods

object-oriented techniques found in literature (see 3.2), or by creating a class which is 'wrapped' around the old sequential subroutine. For Fortran routines this means that the class gets all the necessary data and calls the Fortran routine, which is located in a library, as shown in figure 4.4. This whole setup can even make use of dynamic libraries, thereby avoiding the relinking of the program each time an element type is added or modified. For C functions the same procedure can be used, or the C code can be pasted into the C++ code.

The advantage of this approach is the flexibility to use object-oriented programming techniques to easily develop and implement new element types while maintaining the possibility of creating faster code over more object-oriented code with more overhead, thus longer execution times. The setup also presents the possibility of creating more 'exotic' element

types, as will be shown in 5.2. But most importantly, it allows for the integration of new and old Fortran subroutines while greatly improving the maintainability of the code, due to the locality (encapsulation, see 3.1.1) of data. Specific element types with unique data not needed by other types can store this data in the element type class, without passing through different levels of code, and without the risk of accidental modification by other routines.

4.3 Implementation

The object-oriented structure for a general finite element program has been implemented in the explicit time integration module of B2000, B2ETA. There are multiple reasons for this choice: First of all, the explicit module is an independent part of the B2000 system and could even be seen as a separate program. An explicit finite element program is much less extensive and complex than an implicit finite element program, making it possible to rewrite the code of the entire module in a relatively short period of time. This allows us to demonstrate not only the advantages of the approach for the existing version, but also new possibilities, like the creation of a parallel version, as shown in chapter 8. Also, B2ETA has already been written in C, making the integration in the C++ code easier while still proving the concept, as both C and Fortran are sequential programming languages.

The object-oriented scheme as described in the previous section has been adopted and slightly extended, as shown in figure 4.5. First of all, the module controlling the computation has been split up into a class controlling the proper order of execution of the computations and a class containing the global data controlling the computation itself, like start- and end-time. This is done in order to re-use the general controlling class for other modules and to have an external way of controlling the computation. The other modification is the setup of the material data. The material class creates an array of objects each containing the data of a material number. This allows for the creation of different material data

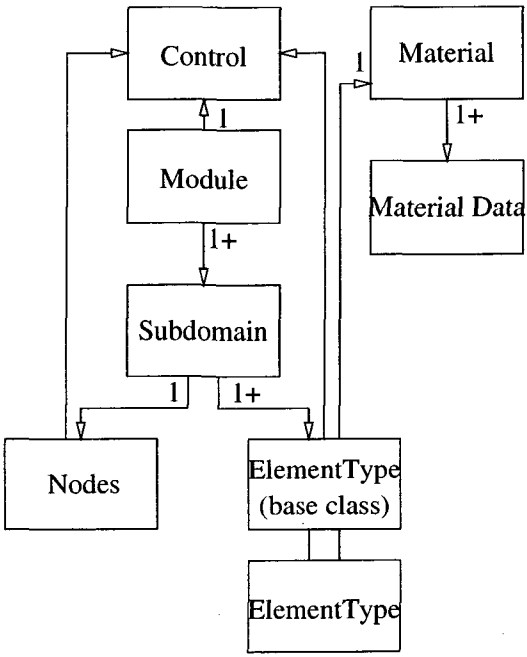


Figure 4.5: Object-oriented scheme of B2ETA

objects, while having one single interface for the rest of the program.

The element base class defines an interface of the element type objects. Its most important functions are `computeFint()` and `computeMass()` in order to compute the contribution to the internal force vector and the mass vector, respectively. The node class stores all nodal data and has the functionality to update them by using `computeV()` and `computeX()`. As the original version was written in C, by far the largest part of the computational code could directly be copied into the C++ code after the new framework had been created.

4.4 Results

The resulting code has been extensively tested and used over the last two years. It has been used for the implementation and testing of new elements (chapter 5), and material models (chapter 6), as well as the implementation of parallelism (chapter 8) and it has been used for new and existing applications (chapter 9). In this section the results of the approach are summarised.

4.4.1 Maintenance and development

The experience with the code as described above shows that debugging has been greatly facilitated thanks to the locality of the data. Because the existing code used in the object-oriented version has been in use for several years before, the validity of the old C code has been well established. The introduction of new features, as described in the following chapters, required a minimum of modifications to the existing code. When during the development and debugging stage errors occurred, they were always to be found in the newly created class. No interference with the existing parts occurred during the extensive development. The object-oriented programming method also allows for some classes extensive checking of the access functionality before introducing them into the existing parts. An example of this code checking is the material data class.

For the introduction of new elements, element specific data are completely encapsulated in the element object, allowing special elements, like the gelatine model described in section 5.2, to be implemented without any modification of the existing code.

In short, based on the experience of the last two years, the conclusion can be drawn that the object-oriented wrapper greatly improves maintenance and facilitates new development.

4.4.2 Performance

One of the important goals set at the beginning of the development of the object-oriented version of the explicit finite element code is that the resulting executable should not be slower than the existing Fortran version. As the old Fortran version has only one isotropic shell element, the possible number and types of tests are limited.

	test 1	test 2	test 3a/b
Number of elements:	20	441	32193
Number of nodes:	42	512	32768
Number of cycles:	160917	4232	186/372

Table 4.1: Summary of performance test cases

Three test cases with a different number of elements were run to compare the performance of the old and the new version of the explicit finite element program. The tests consisted of the vibration analysis of a panel, where the first test consisted of a curved strip and the other two tests of flat panels. The characteristics of the tests are summarised in Table 4.1. Test case three has been run two times, the second time with an integration time twice that of the first run.

test nr.	F77-code time (s)	C++-code time (s)	difference (%)
1	120.	101.	15.8
2	57.0	49.9	12.5
3a	208.	185.	11.1
3b	414.	365.	11.8

Table 4.2: Performance comparison between F77 and C++ code.

The results[1] of the comparison tests between the old Fortran and the new C++ code are shown in Table 4.2 and figure 4.6. The first thing to

[1] Run on a Linux PC, Intel Pentium II-333 MHz. Compile options:
`-O3 -funroll-all-loops -frerun-loop-opt -finline-functions`

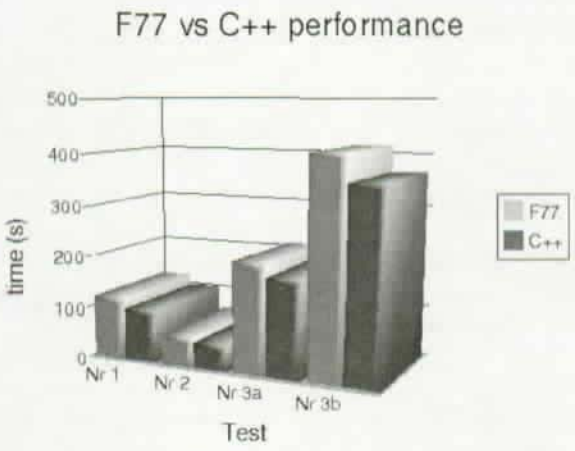


Figure 4.6: Performance comparison between F77 and C++ code.

notice is that the C++ version of the code is faster than the Fortran code. This is due to the fact that the C++ compiler can generate more efficient code for the computational part. In order to facilitate coding a number of small tasks, like the computation of in-products and cross-products of vectors, these functions are written in utility routines. In Fortran this code becomes an object file, which is placed in the library and called from the computing routine. The calling of a different routine destroys the cache line of the processor, as a different function needs to be loaded in the instruction set of the processor. In C++ these small functions are written in an include file, which forces the compiler to inline the code in the actual function. This generates larger object files and executables, but results in more efficient object code. The result is that the C++ version is 10 to 15 percent faster than the old Fortran program.

As a side note on compilers: A compiler directive can request a compiler to try to inline small functions, but does not force it to do so. Fur-

thermore, it must be noticed that the results presented here are somewhat platform dependent. Contrary to the Linux platform, on which these tests were executed, some vendors in the high-performance computing domain, such as NEC and Cray, spent much more effort on their Fortran compiler than on their C++ compiler, which will then show different results in performance than the once presented here.

The other point to notice is the decline in performance gain with increased model size. This is due to the loading phase of the program. The Fortran data is mainly static, so that data allocation is much quicker. Also the initialisation phase and loading of data from database have much less overhead, as everything is loaded directly in the controlling module. The C++ code, however, is fully dynamic and the data from database is loaded in the proper objects, thus generating much more overhead compared to the Fortran version. This can be seen in the difference between test 3a and 3b, where the integration time has been increased to increase the amount of computations with respect to the loading phase. This example clearly shows that with increased computation time the C++ version becomes relatively faster.

So despite the increased overhead generated by the object-oriented programming, the performance of the finite element code is not reduced. Although the overhead requires a longer time for start-up and initialisation, the possibilities of C++, like inline coding, result in more efficient executables and therefore in improved performance of the finite element program.

4.5 Conclusions

In this chapter the object-oriented structure of the explicit finite element program has been described. Contrary to other research and object-oriented finite element programs found in literature, the structure has been designed to preserve the existing Fortran and C computational routines and has been partially based on existing data structures in order to maintain compatibility. Despite the integration with old routines, the

new program structure is object-oriented based, improves maintenance and reduces development time. Existing literature on this subject accepts the increase in execution time as the price to pay for ease of implementation. In this research the goal was set *not* to increase computation time and to reduce it when possible.

The first results shown in section 4.4 demonstrate that both objectives have been met. The resulting program structure allows for the inclusion of traditional sequential programming in Fortran and C, so that the existing code can be re-used in the new code. The object-oriented structure, however, results in better protected data, thus greatly improving the maintenance and ease of development of the code.

The results also demonstrate that the use of an object-oriented programming style does not have to result in longer execution times. As stated in the introduction, a substantial increase of computation time is not acceptable for practical applications and industry demands. The code developed by us showed that the use of C++ can even lead to improved computational times due to the more efficient code generated by the compiler.

5

Implementation of new elements

In this chapter the implementation of two new element types in the explicit finite element code is described. The first element is a standard eight node volume element with one-point integration and hourglass control. This element type has been coded in C as were it to be implemented in the original Fortran program. The C routine has then been integrated in the object-oriented version. The only difference from a Fortran/C implementation is the use of some of the C++ functions written in the include files `b2Util.H` and `b2VecMat.H` which contain utility functions and vector-matrix operations, respectively. The standard eight-node volume element has been implemented to demonstrate the continued possibility to write sequential code without compromising the object-oriented structure.

The second element type described in this chapter is based on the discrete element method (DEM) and has been developed in cooperation with N. Petrinic of the University of Oxford following his ideas. The idea behind the discrete element method is to replace the constitutive relations

of an element with an inter-particle law between the nodes. The element-node connectivity is abandoned and node-node connectivity is computed using a contact-detection algorithm. This kind of element requires a complete different data structure and different information compared to ordinary element types. The implementation based on the discrete element method shows the versatility of the object-oriented code and the advantages of grouping all elements of the same type together in one object.

5.1 A volume element

The 8 node volume element `H8.ETA` is a uniform strain hexahedron as developed by Flanagan and Belytschko and described in [8]. Here follows a brief summary limited to the volume element only.

5.1.1 Kinematics

The isoparametric shape functions map a unit cube in ξ_i space (or explicity written as (ξ, η, ζ)) to a general hexahedron in x_i (or (x, y, z)) space. Choosing the centre of the unit cube as the origin in ξ_i space the shape functions can be written in terms of an orthogonal set of base vectors as follows (see table A.1 for the definition of the vectors):

$$\begin{aligned}\phi_I = &\tfrac{1}{8}\Sigma_I + \tfrac{1}{4}\xi\Lambda_{1I} + \tfrac{1}{4}\eta\Lambda_{2I} + \tfrac{1}{4}\zeta\Lambda_{3I} + \\ &\tfrac{1}{2}\eta\zeta\Gamma_{1I} + \tfrac{1}{2}\zeta\xi\Gamma_{2I} + \tfrac{1}{2}\xi\eta\Gamma_{3I} + \tfrac{1}{2}\xi\eta\zeta\Gamma_{4I}\end{aligned} \tag{5.1}$$

The above vectors represent the displacement modes of the unit cube. The vector Σ_I is the rigid body translation, the vectors Λ_{iI} are the linear, uniform normal strain modes and the last four vectors $\Gamma_{\alpha I}$ can be identified as the linear strain modes which are neglected by the one-point integration. These vectors are called the *hourglass* base vectors as they describe the hourglass patterns for a unit cube. Note that we use the following notation: the capital index I denotes the nodal points and ranges from 1 to 8, the index i ranges from 1 to 3 and the Greek subscript ranges from 1 to 4.

For the relation for the element nodal forces f_{iI} we get, using the principle of virtual work:

$$\dot{u}_{iI}\, f_{iI} = \int_V T_{ij} D_{ij}\, \mathrm{d}V \tag{5.2}$$

where $\dot{u}$ is the velocity, T_{ij} the Cauchy stress tensor, D_{ij} the deformation rate tensor and V the volume. Using one-point integration, thereby neglecting the nonlinear part of the element displacement field, the expression is approximated by:

$$\dot{u}_{iI}\, f_{iI} = V \bar{T}_{ij} \dot{\bar{u}}_{i,j} \tag{5.3}$$

where $\bar{T}_{ij}$ and $\dot{\bar{u}}$ denote the assumed uniform stress field (called the mean stress tensor) and the mean velocity, respectively. The mean kinematic quantities are defined by integrating over the element as follows:

$$\dot{\bar{u}}_{i,j} = \frac{1}{V} \int_V \dot{u}_{i,j}\, \mathrm{d}V \tag{5.4}$$

The B-matrix is now defined as:

$$B_{iI} = \int_V \phi_{I,i}\, \mathrm{d}V \tag{5.5}$$

So that the mean velocity gradient is given by:

$$\dot{\bar{u}}_{i,j} = \frac{1}{V} \dot{u}_{iI} B_{jI} \tag{5.6}$$

Now we can express the nodal forces by:

$$f_{iI} = \bar{T}_{ij} B_{jI} \tag{5.7}$$

Computing the nodal forces now requires evaluation of the B-matrix and the volume. As $x_{i,j} = \delta_{ij}$ and $x_i = x_{iI}\phi_I(\xi, \eta, \zeta)$ we can substitute this into Eq.(5.5) to get:

$$x_{iI} B_{jI} = \int_V (x_{iI}\phi_I)_{,j}\, \mathrm{d}V = V\delta_{ij} \tag{5.8}$$

or

$$B_{iI} = \frac{\partial V}{\partial x_{iI}} \tag{5.9}$$

Using some extensive calculations (see [8]) we can find a way to compute the B matrix and in its turn the volume. These expressions are given in appendix A.4.

5.1.2 Constitutive relations

The only stress-strain relation implemented so far is an isotropic elastic material. Because the integration scheme uses only the linear strain rate terms, the stress rate cannot depend on the nonlinear portion of the displacement field. Hence, the mean stress must be related to the mean strain rates. The physical stress rate for such a material is described by:

$$\dot{T}_{ij} = \bar{T}^{\nabla}_{ij} + \bar{W}_{ik}\bar{T}_{kj} + \bar{W}_{jk}\bar{T}_{ki} \tag{5.10}$$

where $\bar{W}_{ij}$ is the (mean) vorticity tensor and $\bar{T}^{\nabla}_{ij}$ the Jaumann rate. Hooke's law is then defined as:

$$\bar{T}^{\nabla}_{ij} = \lambda \bar{D}_{kk}\delta_{ij} + 2\mu\bar{D}_{ij} \tag{5.11}$$

where $\bar{D}_{ij}$ is the deformation rate tensor and λ and μ the Lamé coefficients. With this constitutive law we can calculate the eigenmodes and the eigenvalues of the hexahedron. According to [3] and [8] the maximum frequency ω_{max} is bounded by:

$$8\,\frac{\lambda + 2\mu}{\rho}\,\frac{B_{iI}B_{iI}}{V^2} \geq \omega^2_{max} \geq \frac{8}{3}\,\frac{\lambda + 2\mu}{\rho}\,\frac{B_{iI}B_{iI}}{V^2} \tag{5.12}$$

where ρ is the density. The central difference time integration scheme is stable if, according to [4, 14]:

$$\Delta t \leq \frac{2}{\omega_{max}}\sqrt{(1+\epsilon^2) - \epsilon} \tag{5.13}$$

where ϵ is the fraction of the critical damping in the highest frequency ($\epsilon < 1$). We obtain for the critical time step for stability of the explicit scheme for the general hexahedron, when omitting the influence of the damping:

$$\Delta t \leq V \sqrt{\frac{\rho}{2(\lambda + 2\mu) B_{iI} B_{iI}}} \tag{5.14}$$

5.1.3 Anti-hourglassing

The constitutive relations combined with the integration scheme described above take only the linear portion of the displacement field into account, causing certain forms of deformation to be energy-free. These are the so-called hourglass modes. In order to prevent instability due to these deformation modes dominating the deformation, the hourglass modes are treated separately.

Considering the total nodal velocity field $\dot{u}_{iI}$ to be made out of the linear portion $\dot{u}_{iI}^{lin}$ and the hourglass field $\dot{u}_{iI}^{hg}$, we can define the hourglass field by:

$$\dot{u}_{iI}^{hg} = \dot{u}_{iI} - \dot{u}_{iI}^{lin} \tag{5.15}$$

It can be shown [8] that the hourglass field is orthogonal to Σ_I and B_{iI} and the base vectors Λ_{iI} except the hourglass base vectors. Therefore, we can expand $\dot{u}_{iI}^{hg}$ as follows:

$$\dot{u}_{iI}^{hg} = \tfrac{1}{\sqrt{8}} \dot{q}_{i\alpha} \Gamma_{\alpha I} \tag{5.16}$$

where the hourglass modal velocities are represented by $\dot{q}_{i\alpha}$ and the constant $\frac{1}{\sqrt{8}}$ is added to normalise $\Gamma_{\alpha I}$. We now define the hourglass shape vector $\gamma_{\alpha I}$ such that:

$$\dot{q}_{i\alpha} = \tfrac{1}{\sqrt{8}} \dot{u}_{iI} \gamma_{\alpha I} \tag{5.17}$$

Using the equations above and following [8] we can compute $\gamma_{\alpha I}$:

$$\gamma_{\alpha I} = \Gamma_{\alpha I} - \frac{1}{V} B_{iI} x_{iJ} \Gamma_{\alpha J} \tag{5.18}$$

Note the difference between the hourglass base vectors $\Gamma_{\alpha I}$ and the hourglass shape vectors $\gamma_{\alpha I}$: For a general hexahedron, $\Gamma_{\alpha I}$ is orthogonal to B_{iI} and defines the hourglass pattern while $\gamma_{\alpha I}$ is orthogonal to the linear velocity field $\dot{u}_{iI}^{lin}$ and is necessary to detect the hourglassing.

There are different possibilities to treat the hourglass difficulties. Most common methods are the addition of artificial damping or the addition of artificial stiffness. The first approach has the disadvantage that hourglass deformation is permanent, since there is no stiffness in the modes. The second approach is shown to be more successful and is incorporated in the element. To control the hourglass modes we define the generalised hourglass $Q_{i\alpha}$ conjugate to $\dot{q}_{i\alpha}$ so that the rate of work is given by:

$$\dot{u}_{iI} f_{iI}^{hg} = Q_{i\alpha} \dot{q}_{i\alpha} \tag{5.19}$$

Using Eq.(5.17) the hourglass nodal force is given by:

$$f_{iI}^{hg} = \frac{1}{\sqrt{8}} Q_{i\alpha} \gamma_{\alpha i} \tag{5.20}$$

Following [8] we get for the hourglass resistance in case of artificial stiffness:

$$\dot{Q}_{i\alpha} = \kappa \frac{\lambda + 2\mu}{3} \frac{B_{iI} B_{iI}}{V} \dot{q}_{i\alpha} \tag{5.21}$$

where κ is the hourglass stiffness parameter. Tests show that a good value for κ is somewhere between 0.01 and 0.05.

5.1.4 Implementation

The theory described above is implemented into the B2ETA program according to the following scheme:

1. Determine matrix B_{iI} according to Appendix A.4
2. Calculate volume V:

$$V = \tfrac{1}{3}(x_I\, B_{1I} + y_I\, B_{2I} + z_I\, B_{3I}) \tag{5.22}$$

3. Calculate mean velocity gradient:

$$\dot{\bar{u}}_{i,j} = \tfrac{1}{V}\dot{u}_{iI}B_{jI} \tag{5.23}$$

4. Calculate mean deformation rate tensor $\bar{D}_{ij}$ and vorticity tensor $\bar{W}_{ij}$:

$$\bar{D}_{ij} = \tfrac{1}{2}(\dot{\bar{u}}_{i,j} + \dot{\bar{u}}_{j,i}) \tag{5.24}$$

$$\bar{W}_{ij} = \tfrac{1}{2}(\dot{\bar{u}}_{i,j} - \dot{\bar{u}}_{j,i}) \tag{5.25}$$

5. Calculate mean stress rate and integrate in time to obtain the new stress:

$$\bar{T}^{\nabla}_{ij} = \lambda \bar{D}_{kk}\delta_{ij} + 2\mu\bar{D}_{ij} \tag{5.26}$$

$$\dot{\bar{T}}_{ij} = \bar{T}^{\nabla}_{ij} + \bar{W}_{ik}\bar{T}_{kj} + \bar{W}_{jk}\bar{T}_{ki} \tag{5.27}$$

$$\bar{T}_{ij} = \bar{T}_{ij} + \dot{\bar{T}}_{ij}\,\Delta t \tag{5.28}$$

6. Calculate internal force vector:

$$f_{iI} = \bar{T}_{ij}B_{jI} \tag{5.29}$$

7. Calculate hourglass forces:

$$f^{HG}_{iI} = \tfrac{1}{\sqrt{8}}Q_{i\alpha}\gamma_{\alpha I} \tag{5.30}$$

where

$$\gamma_{\alpha I} = \Gamma_{\alpha I} - \tfrac{1}{V}B_{iI}x_{iJ}\Gamma_{\alpha J} \tag{5.31}$$

$$\dot{Q}_{i\alpha} = \kappa\frac{\lambda + 2\mu}{3}\frac{B_{iI}B_{iI}}{V}\dot{q}_{i\alpha} \tag{5.32}$$

$$Q_{i\alpha} = Q_{i\alpha} + \dot{Q}_{i\alpha}\,\Delta t \tag{5.33}$$

This scheme is coded in a C routine, which is then called from the element type class in a loop over all the elements of the same type. The element type class is derived from the element base class, which defines the interface and a few basic variables. The integration is as shown in figure 4.4.

5.2 A particle model to simulate gelatine

In structural impact tests, different impactors are used for different types of impact: hard, soft and medium. The hard impactor is usually made of steel, the medium of rubber and the soft impactor is usually gelatine. Apart from the hard impactor, the materials are very difficult to simulate in a numerical analysis, due to the large deformations. Although different models and implementations are available for both the rubber and the gelatine, numerical results are not very satisfactory. In this paper a new approach to the modelling of the gelatine is proposed based on the discrete element method. The model has been implemented in the explicit module of the finite element code B2000 and first test results are presented.

5.2.1 Description of the model

The main purpose of the gelatine is to deposit energy (mass) in a certain way on the impacted structure. In simulations we are not interested in the behaviour of the gelatine, but in the response to the gelatine impact by the structure in question. This lead by N. Petrinic to his idea to simulate the gelatine by particles which get dumped on the surface, instead of trying to solve the stability problems in the existing gelatine models (e.g. imploding elements, etc.). The particles interact with their neighbours by a simple elastic-plastic law (spring-dashpot, as shown in figure 5.1).

The element connectivity is not known in advance, but has to be obtained by a contact search. For this a binary tree search algorithm is used as implemented by N. Petrinic [26].

The contact forces are computed using a penalty method. The penetration p is defined as:

$$p = d - (R_1 + R_2) \tag{5.34}$$

where d is the distance between two particles and R_1 and R_2 the radius of particle one and two, respectively. The definition of the parameters is

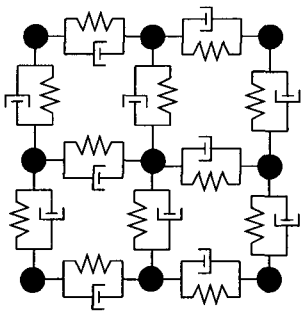


Figure 5.1: Connectivity of nine particles

also shown in Figure 5.2. Now the elastic contact force F_e is computed by a simple law:

$$F_e = k \cdot p \tag{5.35}$$

where k is the penalty stiffness, to be specified in the input. Damping can also be introduced by specifying the damping ratio r, so that the damping force F_d can be computed by:

$$c = r \cdot k \cdot \Delta t \tag{5.36}$$

$$F_d = r \cdot \Delta V \tag{5.37}$$

where c is the damping factor, Δt some critical time step and ΔV the relative velocity between the two particles.

In order to compute the critical time step, the eigenfrequency of two particles is calculated, as shown in Figure 5.3. The eigenfrequency ω and corresponding critical time-step Δt are given by:

$$\omega = \sqrt{k \left(\frac{m_1 + m_2}{m_1 m_2} \right)} \tag{5.38}$$

$$\Delta t = 1/\omega \tag{5.39}$$

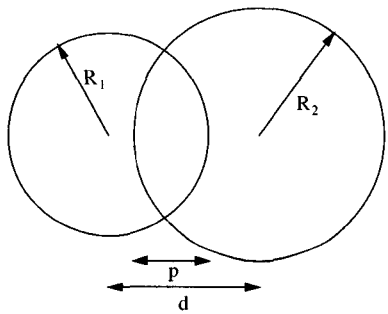


Figure 5.2: Definition of parameters used

So that $t_{min} = 1/\omega_{max}$, where ω_{max} for $m_1 = m_2 = m_{min}$. By taking the minimum particle mass, we need to compute the critical time step only once, thereby having a safe value and reducing computations.

Figure 5.3: Two particles connected by a spring

Additionally, a factor e can be specified which defines the ratio of the radius for which an equilibrium is reached. This allows for the particles to be compressed in their equilibrium position. This modifies Eq.(5.35) to:

$$F_e = k \cdot (p + e \cdot R) \tag{5.35b}$$

Finally, a limited tension contact force is permitted up to a maximum value to allow for a limited tensile strength.

The material parameters must now be obtained by simulating a simple impact test and optimising the material parameters.

5.2.2 Implementation

The implementation of the discrete element method for gelatine shows the flexibility of the setup of the object-oriented structure of the code. Recalling the base class implementation, we know that the base class only provides the interface towards the subdomain and contains some elementary data, like the subdomain number the elements belong to. The main

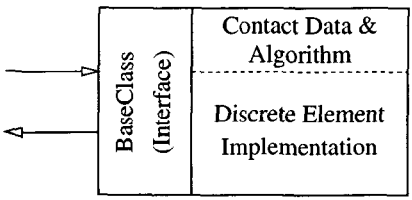


Figure 5.4: Gelatine particle implementation

two interface functions we need to implement are `computeMass()` and `computeFint()`. As the computation of the mass is straightforward, remains the computation of the internal forces. The computation demands only the contribution to the internal force vector. The equations for the interaction between the particles is described above. The data required for the particles are the radius, the material number, their coordinates and the contact search information. The coordinates are obtained from the subdomain (the pointer to the coordinates array is passed), the rest is created within the element type object.

5.2.3 Tests

Apart from some elementary tests to validate the implementation of the particles, a simulation was run to see if the model worked as forseen. For that a small model was made of four particles, slightly compressed and in equilibrium, with an initial velocity V_0. They collide with a fixed wedge, which should split the particles. The model is shown in Figure

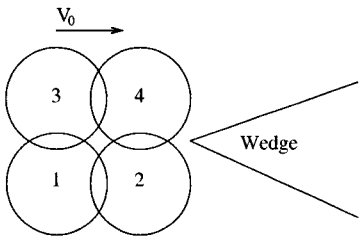


Figure 5.5: Four particles colliding on a wedge

5.5. The trajectories of particle 2 and 4 are shown in Figure 5.6, showing the correct behaviour.

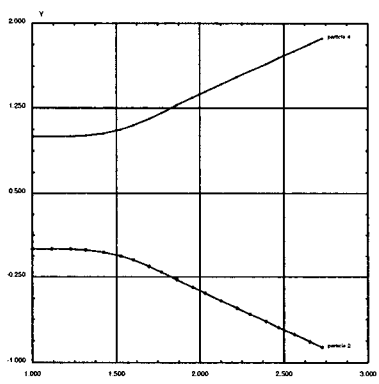


Figure 5.6: Trajectories of particle 2 and 4

5.2.4 Obtaining material parameters

The problem with the discrete element method for gelatine is to find the proper parameters to be used. Instead of ordinary elastic properties for

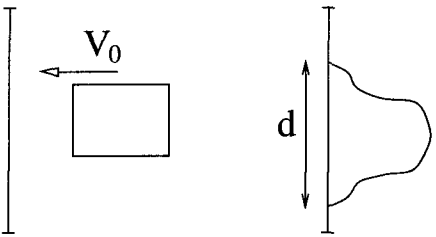


Figure 5.7: Obtaining gelatine parameters: perpendicular impact on rigid wall

continuum elements or hydrodynamic properties for fluid elements, the parameters for the inter-particle law need to be obtained. The best way of doing this is by simulating the impact of gelatine on a rigid wall and modifying the parameters until the actual behaviour of the test is approached. A way of describing the behaviour which can be related to the actual test is to look at the diameter d of the impacted zone in time, as shown in Figure 5.7.

Unfortunately, impact data of gelatine perpendicular to a rigid wall was not available, so that the parameters had to be tuned according to the images and results of the impact tests and simulations as described in section 9.2.

5.2.5 Material behaviour

The simulation of gelatine impact has an additional complication: With increasing speed of the impactor the shape of the gelatine cylinder changes in the test due to inertia, viscosity and aerodynamic effects. (Despite the fact that the impact chamber is pumped vacuum.) This can be seen in Figures 5.8 (a), (b) and (c). These (enhanced) pictures show the shape of the impactor at different velocities. At high velocities the gelatine impactor gets the form of a cone rather than a cylinder. This affects of course the impact on the structure. If the shape of the gelatine impactor is

not changed in the simulations, this will certainly affect the results. But in general the actual shape of the impactor is not precisely known.

(a) *V=145* m/s (b) *V=230 m/s* (c) *V=480 m/s*

Figure 5.8: Gelatine shape at different velocities

However, the existence of these uncertainties does not mean that the results obtained are useless. The purpose of a gelatine model is to deposit mass over a large surface, so that as long as the mass distribution and kinetic energy (i.e. the pressure on the structure) are more or less correct, the resulting response (the damage) is not too sensitive to the change in the parameters of the gelatine model.

5.3 Conclusions

This chapter showed two ways of implementing new elements: by sequential programming wrapped in a class, or by making use of the encapsulation of the element type class to hide the complexities of a special element type. The first method is similar to the way existing sequential routines are integrated in the object-oriented code. The second implementation shows an advantage of grouping elements of the same type together in one element type object.

The new method of modelling gelatine using the discrete element method, according to an idea by N. Petrinic, has been successfully implemented and behaves as expected. Unfortunately, no proper impact data was available in order to calibrate the material properties. In addition, the influence of grain size, behaviour and inter-particle parameters needs to

be investigated. The gelatine model will be used for impact simulations as described in section 9.2 where the inter-particle law parameters are estimated according the behaviour of the gelatine in the impact tests and for a given grain size and density.

6

New material models

In this chapter, the implementation of two new material models is described. They are damage models for composite materials. The study of damage behaviour of composite materials is quite recent. Previously, composite structures were designed in the elastic region, but with increased use of composites in critical parts, the damage behaviour and damage tolerance are becoming increasingly important. The damage behaviour of fibre reinforced composite materials is much more complex than that of metals, due to the anisotropy of the materials themselves and because of the occurrence of different failure modes. The in-ply failure can be fibre failure, matrix failure or a combination of both. The study of inter-ply delamination is an even more complicated task.

The material damage models in this chapter have been developed in the Bright/ Euram project HICAS (High velocity Impact of Composite Aircraft Structures) [15]. Both models are in-plane damage models, omitting the effect of delamination. The theory of the uni-directional (UD) model has been developed by U. Edlund of the University of Linköping, Sweden, the fabric model has been developed by A. Johnson of the German aerospace laboratory (DLR), Stuttgart. The algorithms and implementations are developed by the author. Two possible ways of program-

ming are shown: First the 'classical' Fortran method, then a more object-oriented approach. It shows that the current setup allows for both methods to be used.

6.1 Damage model for UD composite materials

Modern composite materials often come as woven tissue or as sheets with the fibres in one direction. With uni-directional (UD) material, a plate can be built with the fibres of the layers in different directions in order to get the optimum strength for the structure. Although the plate will have the same deformation for all the layers, the same strain will result in different stresses for each layer, depending on the direction of the fibres. For this type of composites a UD material model has been developed by U. Edlund, which describes the damage and failure behaviour of the uni-directional layer. The model is based on the Ladeveze [20] theory of composite damage and has been derived from thermodynamical principles. The model has been described in [7]. An algorithm for numerical implementation has been developed by the author together with U. Edlund and is described here.

In the numerical implementation we have to consider two phases: The identification of the material parameters from some input format and the computation of the material response. The first part is done in the so-called input processor, which reads and interprets data from an input file. Input pertaining the material data can be defined by specifying the material parameters. The input in the B2000 input files is keyword driven. The new material damage model is a plasticity type called `LINK1` and the material data can be introduced by defining the material parameters. The material parameters are then stored on the database. The definition of the keywords is given in Appendix B.1. How the material parameters can be obtained from tests is described in section 9.1.

6.1.1 Algorithm for the UD material

The UD failure model has been developed by U. Edlund as described in [7], which will not be repeated in detail here. In this report only the derived algorithm will be described. First the internal variables are computed, whereafter the damage parameters are integrated in time using a forward Euler scheme. Based on this theory, the following algorithm has been devised:

For the computation we need to store for each time-step in the integration point the total stress ($\boldsymbol{\sigma}$), the strain, ($\boldsymbol{\epsilon}$) and nine additional state variables: the irreversible strain ($\boldsymbol{\epsilon}^{ir}$), the damage variables ($\mathbf{d}$) and the plastic hardening (or softening) variables ($\mathbf{r}$). With these variables the other internal parameters can be computed. Together with the stress rate ($\dot{\boldsymbol{\epsilon}}$) and the time-step ($\Delta\, t$) the new stress state can be computed and the parameters updated.

We start by computing the internal variables at time t_N, denoted by the underscore N. The first are a re-occurring constant Q and the mean effective stress coefficient σ^m:

$$Q(\mathbf{d})_N = 1 - (1 - d_{1_N})(1 - d_{2_N}) \cdot \nu_{12} \cdot \nu_{21} \tag{6.1}$$

$$\begin{aligned} \sigma^m_N = & \tfrac{1}{3} \frac{(1 - d_{1_N})(E_1 + \nu_{12} \cdot (1 - d_2)E_2)}{Q(\mathbf{d_N})} (\epsilon_{11_N} - \epsilon^{ir}_{11_N}) \\ & + \tfrac{1}{3} \frac{(1 - d_{2_N})(E_2 + \nu_{21} \cdot (1 - d_1)E_1)}{Q(\mathbf{d_N})} (\epsilon_{22_N} - \epsilon^{ir}_{22_N}) \end{aligned} \tag{6.2}$$

Based on the sign of σ^m the effective deviatoric stiffness matrix $\mathbf{E}^{Dev}$, the deviatoric stress and the effective stress, $\boldsymbol{\sigma}^d$ and $\boldsymbol{\sigma}^{eff}$, respectively, are computed. If σ^m is positive (i.e. tension), we get for the stiffness

matrix:

$$\mathbf{E}^{Dev} = \begin{bmatrix} \frac{1}{3}\frac{(1-d_{1_N})(2E_1-\nu_{12}(1-d_{2_N})E_2)}{Q(\mathbf{d_N})} & \cdot & \cdot \\ \cdot & -\frac{1}{3}\frac{(1-d_{2_N})(E_2-2\nu_{21}(1-d_{1_N})E_1)}{Q(\mathbf{d_N})} & 0 \\ -\frac{1}{3}\frac{(1-d_{1_N})(E_1-2\nu_{12}(1-d_{2_N})E_2)}{Q(\mathbf{d_N})} & \cdot & \cdot \\ \cdot & \frac{1}{3}\frac{(1-d_{2_N})(2E_2-\nu_{21}(1-d_{1_N})E_1)}{Q(\mathbf{d_N})} & 0 \\ 0 & 0 & (1-d_{12_N})G_{12} \end{bmatrix} \tag{6.3}$$

So that we can compute the deviatoric stress,

$$\boldsymbol{\sigma}_N^d = \mathbf{E}^{dev}\boldsymbol{\epsilon}_N^e = \mathbf{E}^{dev}(\boldsymbol{\epsilon}_N - \boldsymbol{\epsilon}_N^{ir}) \tag{6.4}$$

and the effective stress,

$$\boldsymbol{\sigma}_N^{eff} = \begin{bmatrix} \frac{\sigma^d_{11_N}+\sigma^m(\mathbf{d_N})}{1-d_{1_N}} \\ \frac{\sigma^d_{22_N}+\sigma^m(\mathbf{d_N})}{1-d_{2_N}} \\ \sigma^d_{12_N} \end{bmatrix} \tag{6.5}$$

If the effective mean stress is negative ($\sigma^m < 0$), the cracks are closed and the deviatoric and effective stresses are computed with the damage set to zero ($\mathbf{d} = \mathbf{0}$). The computation is further similar to Eqs.(6.3), (6.4) and (6.5). With the effective stress known, we can compute the plastic flow function:

$$\begin{aligned} f_N = &\sqrt{(\sigma^{eff}_{12_N})^2 + a^2\,(\sigma^{eff}_{22_N})^2} \\ &- (\kappa_0 + \sqrt{(\theta_2\, r_{22_N})^2 + (\theta_{12}\, r_{12_N})^2}) \end{aligned} \tag{6.6}$$

where a is a coupling factor and κ_0 the yield stress.

The first two components of the elastic part of the thermodynamic force vector $\mathbf{Y}$ also depend on the sign of the mean stress. According to

[7]:

$$Y^e_{1_N}(\boldsymbol{\epsilon}^e) = \begin{cases} \frac{1}{2}\left(\frac{\sigma_{11_N}}{1-d_{1_N}}\right)^2 \frac{1}{E_1} & \text{if } \sigma^m \geq 0 \\ Y^-_{1_N}(\boldsymbol{\epsilon}^e) & \text{if } \sigma^m < 0 \end{cases} \tag{6.7}$$

$$Y^e_{2_N}(\boldsymbol{\epsilon}^e) = \begin{cases} \frac{1}{2}\left(\frac{\sigma_{22_N}}{1-d_{2_N}}\right)^2 \frac{1}{E_2} & \text{if } \sigma^m \geq 0 \\ Y^-_{2_N}(\boldsymbol{\epsilon}^e) & \text{if } \sigma^m < 0 \end{cases} \tag{6.8}$$

$$Y^e_{12_N}(\boldsymbol{\epsilon}^e) = \frac{1}{2}\left(\frac{\sigma_{12_N}}{1-d_{12_N}}\right)^2 \frac{1}{G_{12}} \tag{6.9}$$

where Y_1^- and Y_2^- are defined according to Eqs.(51) and (53) in [7]:

$$Y^-_{1_N}(\boldsymbol{\epsilon}^e) = \frac{1}{6}\left[\frac{(2E_1 - \nu_{12}(1-d_{2_N})E_2)\epsilon^e_{11_N}}{Q(\mathbf{d}_N)^2} - \frac{(E_1 - 2\nu_{12}(1-d_{2_N})E_2)\epsilon^e_{22_N}}{Q(\mathbf{d}_N)^2}\right] \cdot (\epsilon^e_{11_N} + \nu_{21})(1-d_{2_N})\epsilon^e_{22_N} \tag{6.10}$$

$$Y^-_{2_N}(\boldsymbol{\epsilon}^e) = \frac{1}{6}\left[\frac{(2E_2 - \nu_{21}(1-d_{1_N})E_1)\epsilon^e_{22_N}}{Q(\mathbf{d}_N)^2} - \frac{(E_2 - 2\nu_{21}(1-d_{1_N})E_1)\epsilon^e_{11_N}}{Q(\mathbf{d}_N)^2}\right] \cdot (\epsilon^e_{22_N} + \nu_{12})(1-d_{1_N})\epsilon^e_{11_N} \tag{6.11}$$

With the elastic part of the thermodynamic forces $\mathbf{Y}$ known, the total force can be computed, using here tensor notation for brevity:

$$Y_{ij_N} = -(Y^e_{ij_N} + \tfrac{1}{2}\theta_{ij} r^2_{ij_N}) \tag{6.12}$$

where $i, j = 1, 2, 3$. Now the damage yield surfaces $\mathbf{g}$ can be computed:

$$g_{1_N} = Y_{1_N} - (l_1 - \xi_1\sigma^m + \beta_1 d_{1_N}) \tag{6.13}$$

$$g_{2_N} = (Y_{2_N} + bY_{12_N}) - (l_2 - \xi_2\sigma^m + \beta_2 d_{2_N}) \tag{6.14}$$

$$g_{12_N} = (Y_{2_N} + bY_{12_N}) - (l_{12} + \beta_{12} d_{12_N}) \tag{6.15}$$

where l_{ij} denotes the damage thresholds and β_{ij} the damage softening parameters.

The last internal variable to be computed before integrating the state variables is the effective plastic state vector $\mathbf{R}^{eff}$:

$$R^{eff}_{ij_N} = \theta_{ij}\, r_{ij_N} \tag{6.16}$$

As we assume that no irreversible strain can take place in the fibre direction, no plasticity can develop in that direction, thus we can set $R^{eff}_{11} \equiv 0$.

With all the necessary internal variables computed, we can integrate the internal state variables $\boldsymbol{\epsilon}^{ir}$, $\mathbf{r}$ and $\mathbf{d}$ using a forward Euler scheme:

$$\epsilon^{ir}_{11_{N+1}} \equiv 0 \tag{6.17}$$

$$\epsilon^{ir}_{22_{N+1}} = \epsilon^{ir}_{22_N} + \frac{1}{\eta_p} \langle f_N \rangle^{m_p} \frac{a^2 \sigma^{eff}_{22_N}}{\sqrt{(\sigma^{eff}_{12_N})^2 + a^2 (\sigma^{eff}_{22_N})^2}} \cdot \Delta t \tag{6.18}$$

$$\epsilon^{ir}_{12_{N+1}} = \epsilon^{ir}_{12_N} + \frac{1}{\eta_p} \langle f_N \rangle^{m_p} \frac{\sigma^{eff}_{12_N}}{\sqrt{(\sigma^{eff}_{12_N})^2 + a^2 (\sigma^{eff}_{22_N})^2}} \cdot \Delta t \tag{6.19}$$

$$r_{11_{N+1}} \equiv 0 \tag{6.20}$$

$$r_{22_{N+1}} = r_{22_N} + \left(-\frac{1}{\eta_p} \langle f_N \rangle^{m_p} \frac{R^{eff}_{2_N}}{|\mathbf{R}^{eff}|} \right) \cdot \Delta t \tag{6.21}$$

$$r_{12_{N+1}} = r_{12_N} + \left(-\frac{1}{\eta_p} \langle f_N \rangle^{m_p} \frac{R^{eff}_{12_N}}{|\mathbf{R}^{eff}|} \right) \cdot \Delta t \tag{6.22}$$

$$d_{11_{N+1}} = d_{1_N} + \frac{1}{\eta_d} \langle g_{1_N} \rangle \cdot \Delta t \tag{6.23}$$

$$d_{22_{N+1}} = d_{2_N} + \frac{1}{\eta_d} \langle g_{2_N} \rangle \cdot \Delta t \tag{6.24}$$

$$d_{12_{N+1}} = b\, d_{2_{N+1}} \tag{6.25}$$

where $\langle\, \rangle$ are the so-called Macaulay brackets, $\langle x \rangle = \frac{1}{2}(x + |x|)$.

To obtain the new stress state we update the total strain,

$$\epsilon_{ij_{N+1}} = \epsilon_{ij_N} + \dot{\epsilon}_{ij_N} \Delta t \tag{6.26}$$

and compute the new internal stress for $t = t_{N+1}$ by computing the deviatoric stress according to Eqs.(6.1) to (6.4). Now the new stress can be computed by

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}^d + \boldsymbol{\sigma}^m \tag{6.27}$$

The substitution of these equations according to [7] yields:

$$\sigma_{11_{N+1}} = \sigma^d_{11_{N+1}} + \langle \sigma^m(\mathbf{d}) \rangle^+ + \langle \sigma^m(\mathbf{0}) \rangle^- \tag{6.28}$$

$$\sigma_{22_{N+1}} = \sigma^d_{22_{N+1}} + \langle \sigma^m(\mathbf{d}) \rangle^+ + \langle \sigma^m(\mathbf{0}) \rangle^- \tag{6.29}$$

$$\sigma_{12_{N+1}} = \sigma^d_{12_{N+1}} \tag{6.30}$$

where $\langle x \rangle^+ = \langle x \rangle$ and $\langle x \rangle^- = \frac{1}{2}(x - |x|)$.

6.1.2 Implementation

The algorithm described above is written as a Fortran subroutine and has been implemented in the B2000 code. The subroutine can also be used, with some minor modifications, as an user material in the finite element code LS-DYNA [12]. This subroutine is then called from the element routine in the class and the corresponding arguments are passed by reference, as Fortran 77 cannot pass a value. The C++ class implementation is passed through the `m4` preprocessor to guarantee portability of the Fortran-C interface. This demonstrates the possibility of continuing to write Fortran code in combination with the object-oriented structure. The inclusion of Fortran code does not influence the setup of the program, nor dies it compromise the object-oriented program structure. Inclusion of Fortran routines for the entire element routine will be similar to the implementation of the material routine.

Some evaluation tests are described in Appendix B.2 where numerical results are compared with some analytical evaluations of the model. As the material damage model is quite complex with numerous of internal variables, relating these to the actual measured stress-strain curves is rather complicated. In order to evaluate the correct behaviour of the implementation, subroutine and single-element tests have been performed and compared with analytical solutions for specific load cases and material parameters. The analytical solutions were obtained by U. Edlund, while the numerical evaluation was done by the author. There were some initial problems for getting some of the Euler schemes started, as well as potential stability problems. The problems of the Euler schemes were solved by the updated model, while the stability requirement turned out to be of no concern when using precise material properties.

As already stated, relating the internal material parameters to the actual behaviour is complicated. Using least-square curve fitting of the quasi-static stress-strain curves to the material model response gives an approximation of some of the parameters. Using these initial guesses, a method for the optimisation of the material parameters by simulating some of the material specimen tests has been developed in a cooperative effort by U. Edlund, N. Petrinic of the university of Oxford, P. Arendsen of the Dutch aerospace laboratory NLR and the author. This optimisation procedure is described in section 9.1.

6.1.3 Updated model

While trying to identify a set of material parameters for the model to fit a set of experimental data, U. Edlund realised that the model described above needed some modifications. The first is related to the extensive plastic damage of the matrix when the ply is loaded in shear. This plastic damage invalidates the coupling between the damage parameter d in the direction perpendicular to the fibres, d_2, and in shear, d_{12} through the material parameter b, as suggested by Ladeveze [20], changing the

expressions for the damage yield functions g_2 and g_{12}, Eq (6.15), to:

$$g_2 = Y_2 - [l_2 - h(\xi_2, \sigma^m) + \beta_2 d_2] \tag{6.31}$$

$$g_{12} = Y_{12} - [l_{12} + \beta_{12} d_{12}] \tag{6.32}$$

This implies for the damage evolution equations

$$\dot{d}_2 = \dot{\mu}_2 \frac{\partial g_2}{\partial Y_2} + \dot{\mu}_{12} \frac{\partial g_{12}}{\partial Y_2} = \dot{\mu}_2 \tag{6.33}$$

$$\dot{d}_{12} = \dot{\mu}_2 \frac{\partial g_2}{\partial Y_{12}} + \dot{\mu}_{12} \frac{\partial g_{12}}{\partial Y_{12}} = \dot{\mu}_{12} \tag{6.34}$$

The second modification concords with the observation that the rate dependent strength seen in experiments seems to require a more general rate dependant damage model. The solution suggested is a two-parameter model similar to model rate-dependent plasticity [33]:

$$\dot{\mu}_1 = \frac{1}{\eta_{d_1}} \langle g_1 \rangle^{m_{d_1}} \tag{6.35}$$

$$\dot{\mu}_2 = \frac{1}{\eta_{d_2}} \langle g_2 \rangle^{m_{d_2}} \tag{6.36}$$

$$\dot{\mu}_{12} = \frac{1}{\eta_{d_{12}}} \langle g_{12} \rangle^{m_{d_{12}}} \tag{6.37}$$

The third modification has to do with the different hardening parameters used in the 12- and 22- direction. The original implementation leads to start problems of the forward Euler scheme used in the algorithm. U. Edlund suggested to use only one hardening parameter for the matrix, which is also reasonable from a physical point of view. Therefore, we set

$$\theta_2 = \theta_{12} = \theta \tag{6.38}$$

This implies:

$$\dot{r}_{22} = \dot{r}_{12} = \dot{r} \tag{6.39}$$

so that we can write for the hardening parameter:

$$\dot{r} = -\dot{\lambda} \frac{-\theta r}{\sqrt{(\theta r)^2 + (\theta r)^2}} = \frac{1}{\sqrt{2}} \dot{\lambda} \tag{6.40}$$

As before, a viscoplastic model is obtained by setting

$$\dot{\lambda} = \frac{1}{\eta_p} \langle f \rangle^{m_p} \tag{6.41}$$

So that we obtain after time discretisation:

$$r_{N+1} = r_N + \frac{1}{\eta_p} \langle f \rangle^{m_p} \frac{1}{\sqrt{2}} \Delta t \tag{6.42}$$

These modifications require some additional material parameters while eliminating others. However, the decoupling makes material parameter identification easier. The material model definition in appendix B.1 includes these modifications.

6.2 Damage model for fabric composite materials

The damage model described in the previous section has been developed for uni-directional fibre composites. The other possibility for composite cloth is to come in a woven fabric. This means that for a ply the fibres are woven in two directions perpendicular to each other. This facilitates the modelling, because the two fibre directions show the same behaviour. The damage model for composite fabric plys described here has been developed by A. Johnson of the German aerospace laboratory (DLR), Stuttgart. Although also based on the Ladeveze theory, Johnson's approach to the development of the material model is quite different with respect to the choice of the damage functions, which are derived directly from the study of the material data, resulting in a much simpler model with a direct relation to the stress-strain curves obtained from tests, as described in [17, 18]. However, it turned out that rate-dependency cannot be introduced properly in this model and has therefore been omitted.

The algorithm described here is implemented in the material class of the explicit module of B2000. The flow diagram of figure 6.1 shows two parts, part *I* and part *II*, where the second part is split up in two, placed before and after part *I*. The first part is the computation of the elastic damage model, as described in subsection 6.2.1, the second part is the extention with plasticity for the shear, as described in subsection 6.2.2.

6.2.1 Elastic damage model

The elastic damage is a specific case automatically applied when the plasticity parameters are set to zero (default). This reduces the amount of computations significantly when plasticity is not required, as is the case for certain forms of impact. The algorithm uses an Euler-like scheme. It follows the equations as found in [16, 17]. Unfortunately, trying to develop a proper Euler scheme would render the equations almost useless due to the complexity of the damage functions, which are linear in the

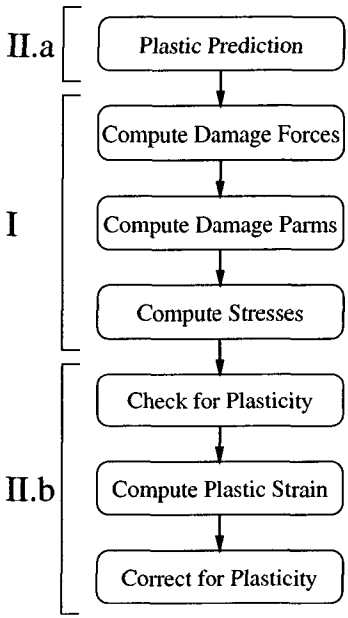


Figure 6.1: Flow diagram of the fabric model

square root of the damage force Y for the axial directions and linear in the log of the square root for the shear. Due to the small time step we expect the procedure to be stable and acceptably accurate.

The same procedure is more or less followed as described for the UD model, but will be repeated here for clarity. Again, first the re-occurring

constant Q and the mean effective stress coefficient σ^m are computed:

$$Q(\mathbf{d})_N = 1 - (1 - d_{1_N})(1 - d_{2_N}) \cdot \nu_{12} \cdot \nu_{21} \tag{6.1}$$

$$\begin{aligned} \sigma_N^m = & \tfrac{1}{3} \frac{(1 - d_{1_N})(E_1 + \nu_{12} \cdot (1 - d_2)E_2)}{Q(\mathbf{d_N})} (\epsilon_{11_N} - \epsilon_{11_N}^{ir}) \\ & + \tfrac{1}{3} \frac{(1 - d_{2_N})(E_2 + \nu_{21} \cdot (1 - d_1)E_1)}{Q(\mathbf{d_N})} (\epsilon_{22_N} - \epsilon_{22_N}^{ir}) \end{aligned} \tag{6.2}$$

where the irreversible strains ϵ_{ij}^{ir} are zero (and are omitted in the code). Based on the sign of σ^m the constants[1] α_1, Y_1^0 and Y_1^f are chosen as being the tension or compression values.

Then some intermediate or estimated stress state is computed based on the damage parameters of step N and the current strains for step $N+1$:

$$\tilde{\boldsymbol{\sigma}}_{N+1} = \mathbf{E}(\mathbf{d}_N)\, \boldsymbol{\epsilon}_{N+1} \tag{6.43}$$

Based on these values for the stress the conjugate damage forces are computed:

$$Y_{1_{N+1}} = \frac{(\tilde{\sigma}_{11})^2}{2E_1(1 - d_{1_N})^2} \tag{6.44}$$

$$Y_{2_{N+1}} = \frac{(\tilde{\sigma}_{22})^2}{2E_1(1 - d_{2_N})^2} \tag{6.45}$$

$$Y_{12_{N+1}} = \frac{(\tilde{\sigma}_{12})^2}{2G_{12}(1 - d_{12_N})^2} \tag{6.46}$$

Based on these forces the new damage parameter values can be computed

[1] Note that a slightly different notation for the constants is used than in [16]. The second underscore in this report is used for the notation of the integration cycle N.

according to [16], Eqs.15:

$$d_{1_{N+1}} = \begin{cases} 0 & \text{if } \underline{Y}_1 \leq Y_1^0 \\ \alpha_1(\underline{Y}_{1_{N+1}} - Y_1^0) & \text{if } Y_1^0 < \underline{Y}_1 < Y_1^f \\ 1 & \text{if } \underline{Y}_1 \geq Y_1^f \end{cases} \tag{6.47}$$

$$d_{2_{N+1}} = \begin{cases} 0 & \text{if } \underline{Y}_2 \leq Y_1^0 \\ \alpha_1(\underline{Y}_{2_{N+1}} - Y_1^0) & \text{if } Y_1^0 < \underline{Y}_2 < Y_1^f \\ 1 & \text{if } \underline{Y}_2 \geq Y_1^f \end{cases} \tag{6.48}$$

$$d_{12_{N+1}} = \begin{cases} 0 & \text{if } \underline{Y}_{12} \leq Y_{12}^0 \\ \alpha_1(\ln(\underline{Y}_{12_{N+1}}) - \ln(Y_{12}^0)) & \text{if } Y_{12}^0 < \underline{Y}_{12} < Y_{12}^f \\ 1 & \text{if } \underline{Y}_{12} \geq Y_{12}^f \end{cases} \tag{6.49}$$

where, according to [16], Eqs.13:

$$\underline{Y}_i(t) = \max(\sqrt{Y_i(\tau)}), \quad \tau \leq t \tag{6.50}$$

where $i = 1, 2, 12$. With the new value of the damage parameters $d_{i_{N+1}}$ the final stress can be computed after computing the new value for $Q(\mathbf{d})$:

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} \frac{(1-d_{1_{N+1}})E_1}{Q} & \cdot & \cdot \\ \cdot \quad \frac{\nu_{12}(1-d_{1_{N+1}})(1-d_{1_{N+1}})E_1}{Q} & & 0 \\ \frac{\nu_{12}(1-d_{1_{N+1}})(1-d_{1_{N+1}})E_2}{Q} & \cdot & \cdot \\ \cdot \quad \frac{(1-d_{2_{N+1}})E_2}{Q} & & 0 \\ 0 & 0 & (1-d_{12_{N+1}})G_{12} \end{bmatrix} \begin{bmatrix} \epsilon_{11_{N+1}} \\ \epsilon_{22_{N+1}} \\ \epsilon_{12_{N+1}} \end{bmatrix} \tag{6.51}$$

6.2.2 Plastic damage model

As described in [16], the in-plane shear deformations are controlled by the behaviour of the matrix, which deforms plastically. This irreversible

strain can be shown by unloading after damage. This leads to the introduction of plastic deformation in the damage model for in-plane shear.

The algorithm implemented in B2000 is as follows: First, a prediction of the elastic strain is made based on the previous value of the plastic strain, if plastic deformation has already occurred:

$$\tilde{\epsilon}^{e}_{12_{N+1}} = \epsilon^{tot}_{12_{N+1}} - \epsilon^{p}_{12_N} \tag{6.52}$$

With the predicted value for the elastic strain, $\tilde{\epsilon}_{12}$, the elastic damage is computed as described in subsection 6.2.1. After that, a check is made to see if plastic deformation has taken place by computing the hardening function $R(p)$:

$$R_{N+1} = \frac{\sigma_{12_{N+1}}}{1 - d_{12_{N+1}}} - R_0 \tag{6.53}$$

where R_0 is the plastic threshold stress. If the hardening function R is less than zero, no plastic deformation has occurred and the real elastic strain is equal to the prediction. Otherwise, the plastic increment and the correct elastic shear strain are computed according to:

$$p_{N+1} = \left(\frac{R}{\beta}\right)^{1/m} \tag{6.54}$$

$$\Delta p = p_{N+1} - p_N \tag{6.55}$$

$$\Delta\epsilon^{p}_{12} = \frac{\Delta p}{1 - d_{12_{N+1}}} \tag{6.56}$$

$$\epsilon^{p}_{12_{N+1}} = \epsilon^{p}_{12_N} + \Delta\epsilon^{p}_{12} \tag{6.57}$$

$$\epsilon^{e}_{12_{N+1}} = \epsilon^{tot}_{12_{N+1}} - \epsilon^{p}_{12_{N+1}} \tag{6.58}$$

Now with the corrected value for the elastic shear strain, the elastic damage is recomputed following subsection 6.2.1.

6.2.3 Implementation

The fabric damage model is implemented as a function of the material class. The current implementation is in the class `b2Material`, of which

only one is present per process and through which the corresponding material data object is accessed. This way the element class does not have to access all the material data, leaving the data encapsulated in the material data object. One could even envisage a specialised object for each material type with the corresponding functionality in this specialised class. This method would allow for complete encapsulation of the material data, but proper implementation is complicated. First of all, if a standard interface is required for all material type classes, a virtual base class is requires for the material class. As the amount of computations on the material data is mostly limited, the effect of a double function affecting the performance is not negligible. For the element class, where this construction is used, this is of no concern, as the amount of computations within the element class is very large. Secondly, the material damage models are considered like plasticity, an extention to the more general orthotropic material. Either this is implemented by a double inheritance scheme, reinforcing the penalty described above, or the different damage models are all implemented in the same material class. This not only complicates the material data class and reintroduces the `if` statements, which we can eliminate with the inheritance scheme, it will also increase the memory use substantially when multiple orthotropic materials are defined.

6.2.4 Material parameters and validation

As the material model is derived directly from studying the stress-strain curve, obtaining the material parameters is relatively simple. By introducing the material law equations in a spreadsheet which also contains the stress-strain data, a least-square method can be used to obtain the material parameters. Also, the limited number of parameters used in this model greatly facilitates the task.

Once the material parameters are obtained, the validation consists of simulating the simple tension and shear tests of the quasi-static material tests. These simulations have been performed with an one-element

model. The results of these one-element simulations are shown in appendix B.4.

6.3 Conclusions

In this chapter two new material models for damage in a composite ply have been described. The models have been developed by U. Edlund and A. Johnson. The algorithms, implementation and validation have been performed by the author. The UD material model has been implemented as as Fortran subroutine in such a way that it can also be used as a user-defined material in the finite element program LS-DYNA3D. The fabric model has been implemented as a function of the general material class, demonstrating the flexibility of implementation of the finite element program structure.

The two material damage models, although both based on the Ladeveze theory for the damage modelling of plys, are developed out of two different principles. The first UD-material model has been developed from a scientific/mathematical approach, where first the basic equations are established based on continuum mechanics. Afterwards the model is modified for better correspondence with the material test data. In order to validate this model, use is made of analytical solutions. The advantage of this approach is the scientific correctness of the model and the availability of exact solutions with which to compare the numerical results. This gives a good idea of the theoretical behaviour of the model and facilitates evaluating the numerical implementation. The disadvantage is, however, that this method does not give a clue about the real behaviour of the material. In addition, the large number of material parameters required constitutes an important problem, only to be solved with the application of an optimisation procedure.

The second fabric model has been developed with an approach nearer to engineering, i.e. it is based directly on the study of material data. However, this method results in some strange relationship between the internal damage parameters, which makes the derived rate-equations impossible

to implement. The simplicity of the model makes for faster computation and gives a quick understanding of the material behaviour. The close relation to the material test results facilitates the computation of the internal material parameters. With the material test data and the equations of the model in a spreadsheet, the internal parameters can be found by using the available least-square method of the spreadsheet. This simplicity also has its drawbacks, making it less able to approximate the real behaviour by optimising the material parameters.

7

Introduction to parallel programming

The continuous development of the finite element method allows the engineer to solve problems of increasing complexity. Although the development of the microprocessor goes at an incredible pace[1], today's demands for computational power are growing faster than the computer industry can deliver. Computational structural mechanics is going the same way as computational fluid mechanics or physics, trying to solve problems of such size and complexity that even on todays computers, computations take hours or even days to complete. Some of the problems are even too large for all but the largest supercomputer due to memory, disk and/or computing demands. But supercomputers are extremely expensive compared to modern workstations and personal computers. So in order to reduce computation time and to allow for large problems to be solved, parallel computing provides an alternative solution.

The idea of parallel computing is relatively simple: If you take two

[1] Moore's law states that the speed of a processor doubles every 18 months, which has indeed been the case for the last 20 years

processors instead of one to do the same task, each solving a part of the problem simultaneously, you could do the task twice as fast. Thus, by making use of multiple processors at the same time, you can greatly reduce the total computation time[2]. Also, by using more computers for the same task, you can make use of the resources of each machine, thereby increasing the amount of memory and disk space available.

This chapter will give a brief overview on parallel programming. First of all the reasons for and limitations of the use of parallel computations are dealt with in section 7.1. Since the chosen methods are inherently linked to the targeted platforms an introduction of some hardware will be given in section 7.2. Section 7.3 then concludes with an overview of paradigms and methods used to create parallel programs. The theory in this section is by no means complete or even extensive. For a more thorough overview see the excellent book by Almasi and Gottlieb [1].

7.1 Amdahl's law

If we could write a perfectly parallel program (i.e. 100% parallel), then the time to execute this program would be, assuming all processors are the same:

$$T_p = \frac{T_s}{P} \tag{7.1}$$

where T_p is the execution time in parallel, T_s the execution time in serial and P the number of processors. Apart from some very specific tasks, like Monte Carlo simulations, we can never write a program which can be fully parallelised. There are certain parts which can only be executed in serial, thereby limiting the maximum speedup possible, where speedup is defined as the quotient of the time for serial and parallel execution:

$$\text{speedup } S_p = \frac{T_s}{T_p} \tag{7.2}$$

[2]The actual time waiting for a computation to finish is called wall-time, the time spent looking at the wall waiting for the computation to complete.

This results in Amdahl's law (Gene Amdahl, 1967):

> If S is the fraction of a calculation that is serial and $1-S$ the fraction that can be parallelised, then the greatest speedup that can be achieved using P processors is: $\frac{1}{(S+(1-S)/P)}$ which has a limiting value of $1/S$ for an infinite number of processors.

Although this sounds trivial, the consequences are clear. If a program has a serial component of 10%, then the maximum speedup possible is 10. The reality is often even worse, as we will see.

The parallel execution time can be split up in multiple terms:

$$T_p = Tcomp_p + Tcomm_p + T_{overhead} + other\ terms \tag{7.3}$$

where $Tcomp_p$ is the parallel computation time, $Tcomm_p$ the time spend in communication and $T_{overhead}$ additional operations for parallel computations. It can be that in order for a problem to be solved in parallel, additional computations need to be performed, or the program might even require a completely different algorithm which can be parallelised. Leaving the overhead and other possible terms aside, the speedup for a perfectly parallel program (i.e. $S = 0$) can now be written as:

$$\frac{1}{S_p} = \frac{T_p}{T_s} = \frac{1}{P} + \frac{Tcomm_P}{T_s} \tag{7.4}$$

We now introduce the computation to communication ratio r_p:

$$r_p = \frac{T_s}{Tcomm_P} \tag{7.5}$$

which can be seen as a typical number. The bigger this ratio the better. This gives for the speedup:

$$\frac{1}{S_p} = \frac{1}{P} + \frac{1}{r_p} \tag{7.6}$$

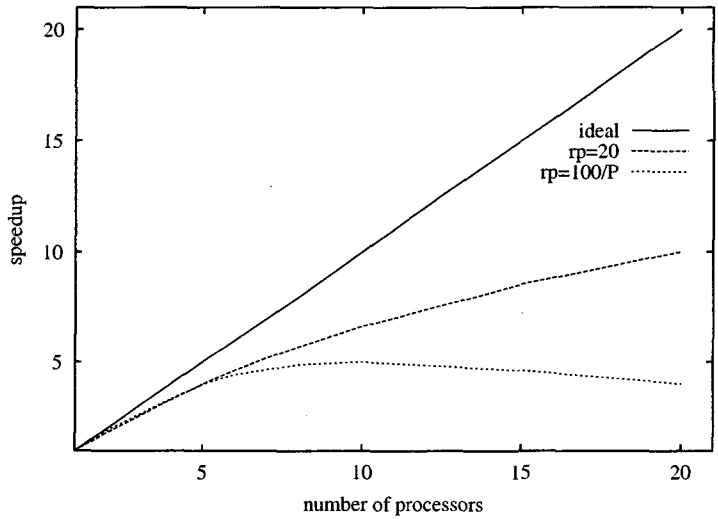


Figure 7.1: Influence of r_p to the speedup

The influence of this term to the speedup is shown in Figure 7.1. The line labelled "ideal" is the ideal speedup, the two other lines show two different behaviours of the parameter r_p. The graph labelled "rp=20" is the speedup for r_p constant with a value of 20. This is an application where the total time spent in communication remains constant with an increasing number of processes. A value of 20 means that 5% of the time compared to the serial time is used for communication. This means for the maximum speedup possible:

$$S_p \rightarrow r_p \quad \text{for} \quad P \rightarrow \infty \tag{7.7}$$

The graph labelled "rp=100/P" is a case where the ratio of communication to computation increases linearly with the number of processes. We see that a maximum speedup is reached before the communication

overhead starts to dominate, effectively reducing the speedup. For an algorithm where the same amount of information needs to be sent from one processor to all processors we can write for r_p:

$$r_p = \frac{T_s}{P \cdot Tcomm} = \frac{r_{p_1}}{P} \tag{7.8}$$

To obtain the maximum speedup possible we need $\frac{d(1/S_p)}{dP} = 0$, so that we can write for this case:

$$\frac{1}{P^2} = \frac{1}{r_{p_1}} \tag{7.9}$$

and the maximum speedup becomes:

$$S_{p_{max}} = \frac{1}{2}\sqrt{r_{p_1}} \tag{7.10}$$

So for the case with $r_{p_1} = 100$ we have a maximum speedup $S_p = 5$ for $P = 10$.

In order to find values and expressions for r_p we have to study not only the algorithm but also the hardware. For the calculation, the time spent in serial computation can be written as:

$$T_s = \frac{a}{V_p} \tag{7.11}$$

where a is the number of million floating point operations (MFLOP) for the application, and V_p the nominal processor speed obtained in MFLOP/s. The processor speed obtained depends on the hardware specifications of the computer, on the program itself and how it is written and on the performance of the compiler[3].

[3] As an example of the performance of a compiler, tests show a five to six fold improvement in CPU time on the ALPHA processor by using the native compiler from Compaq instead of the GNU compiler. This performance increase is obtained by using the processor more efficiently by rearranging operations and by reducing the number of operations by code optimisation by the compiler itself.

The communication depends on the communication scheme, the amount of data to be transferred, and the communication hardware. For each communication between two processes the time spent on communication can be written as:

$$T_c = \frac{b}{B} + L \tag{7.12}$$

where b is the amount of data to be transferred, B the bandwidth of the network in Mb/s and L the latency, the time needed to set up the communication. Applications which send small packets of information are usually limited by the latency of the network, while applications sending large amounts of data to only a few processes are usually bound by the bandwidth.

For each parallel program and algorithm we can now perform an analysis with increasing complexity of the parallel performance of the type described above. Although the actual performance will not be exactly the same because the actual implementation will in general be more complicated than can be modelled and is very much dependent on the hardware, the analysis will give an good impression of the possible gains of parallel computing for the application.

7.2 Hardware types

Linked to the theory of parallel computing is the computer itself. The hardware is an important factor for how parallelism is achieved. The same parallel program written with a specific architecture in mind will perform differently (or perhaps not at all) on a different computer. In this section we will briefly discuss some of these differences.

7.2.1 Process streams

A common way of how parallel architectures are conceived is by looking at the execution of the instruction stream. The classical model of a

processor is the Von Neumann model. This model assumes that the processor fetches data from memory and performs an operation on it. This is the classical way of looking at a computer. Reality is usually more complicated, but the model helps in understanding the workings of a processor. This model is of a single operation on a single piece of data and is therefore referred to as *SISD* (single instruction single data).

Based on this model one can describe the methods of archiving parallelism in a computer. If the same operation can be simultaneously performed on another piece of data, we get a *SIMD* computer (single instruction multiple data). This model is used in vector computers, where the same operation is performed on a large array of data. This kind of parallelism can be extremely efficient, and some of the fastest super computers are vector machines. But this works only for a specific type of problem with large arrays, as can be found in scientific applications with vector and matrix operations.

Extending the idea of performing a single operation on multiple data, one can perform different operations on different data at the same time. This *MIMD* (multiple instructions multiple data) is the most common form of parallel computing in use today, because such a computer architecture allows for the most flexibility. Multiple processors are connected by some form of network, which requires some method of communication between the processes.

The other possibility in this scheme, *MISD* (multiple instruction single data), where one could imagine performing different operations on the same piece of data, has never been implemented.

7.2.2 Memory models

The data for a computation has to be stored in memory. For this there exist two different architectures, each with its own advantages and disadvantages: Shared-memory and distributed-memory machines. A scheme of both categories is given in Figure 7.2.

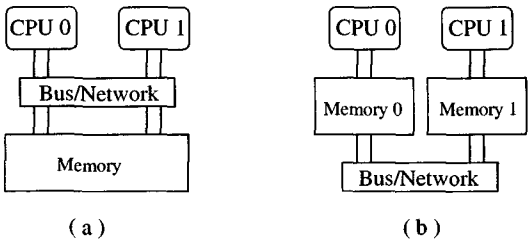


Figure 7.2: Shared (a) and distributed (b) memory

Shared Memory

In a parallel computer with a shared memory architecture, all processors have access to the same, and therefore "shared", memory. This means that a given processor does not need to know the location of the data needed with respect to the other processors, since all data is stored in this "shared" memory. In general, a computer with a shared memory architecture is therefore easier to program than in distributed counterpart. A disadvantage of these types of machines is that access to the shared memory "pool" is often slow, particularly for computers with large amounts of memory. This is because, in general, the physical location of the memory is relatively far from the processor. All access to the memory has to pass through the same bus, making the bandwidth of this bus a bottleneck in memory access.

Programming shared memory allows for easy access by all processors to the same memory, while creating, however, its own problems. Read access of multiple processes to the same part in memory can be performed asynchronously, write operations, however, must be executed by one processor at the time (synchronous access). Safeguard mechanisms must be implemented in the code in order to prevent accidental overwriting of shared data.

Distributed Memory

In a distributed architecture each processor has its own local, and therefore fast, memory. However, this speed has a price, namely that the program, and therefore the programmer, has to know which data has to be transferred to which processor, thus demanding a considerable effort and making distributed memory machines harder to program than the above mentioned shared memory computers. But distributed memory computers are in general much cheaper, because no efficient bus to the memory is required. They can even be constructed by connecting multiple machines over a network. However, the communication between two processors is not only more complicated, but in general also much slower.

The trend in parallel computers is twofold: machines with a relatively small number of processors (1, 2 or 4) on the desktop using shared memory, and clustering relatively cheap desktop machines together to create a large parallel computing environment. Multi-processor servers, like the SGI Origin series, use a mixed approach with software making it look like a shared memory machine. But the number of processors to be connected is limited, making the machine less scalable. On the other hand, with the low prices for personal computers and improving networks, clustering machines together from off-the-shelf components for specific computing purposes, like the Beowulf type class machines[4], is becoming very attractive for their price/performance ratio.

[4]The term Beowulf comes from the NASA Goddard Space Flight Center, where the first computer of this type was build. By definition and philosophy it is based on COTS (Commodity off the shelf) components and all system software required to construct and operate a Beowulf should be publicly available. A Beowulf class cluster computer is distinguished from a Network of Workstations (NOW) by several subtle but significant characteristics. First, the nodes in the cluster are dedicated to the cluster. Also, since the interconnection network is isolated from the external network, the network load is determined only by the application being run on the cluster. Finally, operating system parameters can be tuned to improve performance.

7.3 Basic parallel methods

Apart from the hardware-parallel models discussed above, there is also a number of software programming models related to the use of parallel computers. In general there are two methods for decomposing a problem in order to make it suitable for parallel execution: A "functional" decomposition method and a "data" decomposition method. The best method depends very much on the type of problem and/or program.

The first method tries to analyse a problem by basing it on the different functions which are needed to solve the problem. For instance, in a weather simulation it is possible to decompose the problem along the lines of the interfaces between land, air, and water. Each of these three partitions (land, air, and water) require a different solution method, hence the name *functional decomposition.*

The second method tries to solve the problem by decomposing the data which is used during the calculation, in other words, each processor uses the same code on a different portion of the data set. This method is frequently used in computational engineering since, in general, its performance with respect to scalability is better than the functional approach. Of course a combined approach is also used, e.g. in the field of multidisciplinary simulations like fluid- structure interaction.

Another way of classifying parallel programs is by following the same idea of parallelism as described in subsection 7.2.1. This gives us two practical implementations: *SPMP* (single program multiple data) and *MPMD* (multiple programs multiple data). The first type is the one often used in scientific computations: A program is copied to different processors and performs the same kind of operations on different parts of the data. The second type is preferred for applications performing different tasks, like a postprocessor, where the data extraction, the operations on the data and the viewer can be implemented as different programs working closely together.

8

Parallelisation of the explicit code

This chapter describes the parallelisation of the explicit finite element code, its implementation and performance. First the method used for parallelising the explicit code, domain decomposition, is discussed. The theoretical performance of the domain decomposition is analysed in section 8.2 using the theory based on Amdahl's law of chapter 7. Then the implementation of the communication and the synchronisation of the computation in the object-oriented code are described. After the description and discussion of some evaluation tests in section 8.4, conclusions are drawn regarding the possibilities and performance of the parallel explicit finite element method.

8.1 Domain decomposition

As discussed in section 7.3, there are basically two methods of parallelising a program, by functional decomposition or by data decomposition. For the explicit finite element method the choice of decomposition

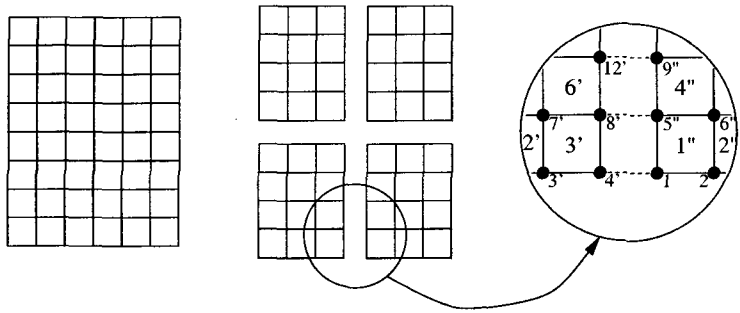


Figure 8.1: Mesh decomposition of a rectangular shell

is rather straightforward, following the theory in section 2.2. The equations of motion have been decoupled, so that they can already be solved in parallel. The remaining problem is the integration over the elements. The computation of the stresses is independent, but the computation of the nodal forces is coupled, as each node gets its contribution from each connected element.

The method now used is called mesh decomposition, where the finite element model gets split up into multiple subdomains as shown in Figure 8.1. Each process now has one or more subdomains over which it performs its integration. The connecting nodes over which the mesh is split get copied to each subdomain. A connectivity matrix has to be constructed between all the subdomains. The nodes in this matrix need to obtain not only the contribution of the connected elements in the subdomain, but also the contribution from the other subdomain. Thus, we can write for the nodes $4'$ and $1''$ from subdomain 1 and 2, respectively:

$$\mathbf{F}^{4'}_{int_{total}} = \mathbf{F}^{1''}_{int_{total}} = \mathbf{F}^{4'}_{int} + \mathbf{F}^{1''}_{int} \tag{8.1}$$

The same goes for the computation of the nodal masses, if required, as the

nodes should obtain the contribution of the mass of the adjacent elements:

$$\mathbf{M}^{4'}_{int_{total}} = \mathbf{M}^{1"}_{int_{total}} = \mathbf{M}^{4'}_{int} + \mathbf{M}^{1"}_{int} \tag{8.2}$$

With the forces and masses on both nodes now equal, the time integration scheme will guarantee the motion of both nodes to be the same, thereby ensuring structural integrity and the correctness of the time integration for the entire structure, assuming that the time step is the same for all subdomains. For this purpose the time step needs to be synchronised over all subdomains such that

$$\Delta t_{global} = \min(\Delta t_i) \qquad i = 1, \text{number of subdomains} \tag{8.3}$$

This way the parallelisation is complete. The problems which now remain are the decomposition of the mesh and the distribution of the subdomains for a well balanced computation. In B2000 the mesh decomposition is often already part of the model, as the total geometry of a model is given by creating mesh patches. For programs or preprocessors which do not have an built-in mesh decomposition capability, mesh partitioners are available. These split the mesh based on graph theory in order to create decomposed meshes, either optimised for minimum connectivity size, equal number of nodes or equal number of elements. However, the theory of mesh decomposition is outside the scope of this thesis.

The distribution of the subdomains over the processors is static, which means that once the subdomains have been distributed they remain where they are. Dynamic schemes have been developed where subdomains can change from one process to the other during the computation to balance the load. This has not been implemented due to its complexity. Also it will not give any better insight in the possible gains when well balanced subdomains are used for the benchmarks.

8.2 Performance analysis

Using the domain decomposition, in principle almost complete parallelisation can be obtained assuming that the domains are well balanced. With

this assumption we can perform an analysis of the parallel behaviour of the code according to the theory described in section 7.1.

Recalling the time spent in serial computation:

$$T_s = \frac{a}{V_p} \tag{7.11}$$

where here a is the number of million floating point operations per cycle. The factor a is problem dependent. Tests have shown the explicit code to perform at about 25% of the peak performance for scalar processors.

The communication time is dependent on the amount of information sent to each processor and on the number of connections required. For the domain decomposition the communication time also depends on the way the mesh is decomposed. For the performance of the explicit code here we consider two scenarios. First we will assume that the mesh is cut in only one direction, so that each domain has only two adjacent domains and that, consequently, communication is only required with maximum two processes containing the adjacent part of the structure. This means that the ratio of communication r_p is constant:

$$Tcomm_p = 2\left(\frac{b}{B} + L\right) \tag{8.4}$$

Then, according to Eq.(7.7) the maximum speedup is given by

$$S_{p_{max}} = r_p = \frac{a/V_p}{2(\frac{b}{B} + L)} \tag{8.5}$$

As an application example we take a 100x100 mesh of shell elements. Analysis and measurements give approximately the values for the code as shown in Table 8.1 We now get for the number of floating point operations per cycle:

$$a = \frac{2000 \cdot 100^2 + 90 \cdot 101^2}{1e6} = 20.9\,\text{MFLOP/cycle} \tag{8.6}$$

and for the amount of data to be transferred (101 connectivity nodes, two-way):

$$b = \frac{2 \cdot 101 \cdot 96}{1e6} = 0.02\,\text{Mb/cycle} \tag{8.7}$$

The performance is of course machine and network dependent. To demonstrate this we compare the performance on a cluster of ALPHA machines (peak performance 1GFLOP/s) and on a cluster of Intel based PCs (peak performance 200 MFLOP/s) both using Fast Ethernet as network connection (10Mb/s, latency 500 μs).

$$S_{p_{alpha}} = \frac{20.9/250}{2((0.02/10) + 0.0005)} = 16.7 \tag{8.8}$$

$$S_{p_{intel}} = \frac{20.9/50}{2((0.02/10) + 0.0005)} = 83.5 \tag{8.9}$$

It is interesting to see what happens if we have an infinite bandwidth or infinitely small latency. The speedups possible on the ALPHA with $B \to \infty$ or $L \to 0$ are:

$$S_{B\to\infty} = 83.6 \tag{8.10}$$

$$S_{L\to 0} = 20.9 \tag{8.11}$$

We see that with this way of decomposing the mesh, the speedup is mainly determined by the network bandwidth.

As a second example, we could decompose the mesh as shown in figure 8.1. The mesh is decomposed in such a way that each domain is connected to the other domains. The number of nodes connected remains

FLOP/element	FLOP/node	comm. byte/node
2000	90	96

Table 8.1: Number of operations for shell element

approximately the same, but the number of connections increases linearly with the number of processes. This way we get for the communication time:

$$Tcomm_P = \left(\frac{b/B}{P} + L\right) P \tag{8.12}$$

Although this does not hold for a decomposition of more than 4 domains, it will give us a worst-case scenario. This way we get for the maximum number of processes:

$$P = \sqrt{\frac{a}{V\,L}} \tag{8.13}$$

We now see that in this case the explicit code is limited by latency and not by the bandwidth of the network. Taking the numbers from above, we get for the maximum number of processes and the corresponding speedup:

$$P_{max_{alpha}} = 13 \qquad S_{p_{alpha}} = 5.6 \tag{8.14}$$

$$P_{max_{intel}} = 29 \qquad S_{p_{intel}} = 10.3 \tag{8.15}$$

The analysis shows that the explicit code in principle is very well suited for parallelisation even for relatively small problems, but the performance is largely influenced by the decomposition of the mesh.

8.3 Implementation

The implementation of parallelism into the explicit finite element code is made easier by the object-oriented code. Recalling figure 4.5 we see that the program already contains a subdomain class. The subdomain object contains the entire mesh, as well as additional data and functions to perform on the subdomain in question. Assuming the domain decomposition is already included in the input (as is the case in B2000), then multiple

subdomains are simply included by creating the required number of subdomain objects. All that remains to implement is the communication between these subdomains. The library used for this communication, MPI, is described in subsection 8.3.1. For the program to run in parallel two new things have to be implemented, one for communication and one for control. The structure of these modifications is described in subsection 8.3.2. The method of communication between and synchronisation of the subdomains is described in subsection 8.3.3 and 8.3.4. Finally the access to data on disk is discussed in subsection 8.3.5

8.3.1 Message Passing Interface (MPI)

One of the problems related to parallel implementations of a program is the communication between the processors and the portability of its implementation to different platforms. Initially, several communication libraries were available, with PVM (Parallel Virtual Machine) being the most well known. In 1992 a message passing interface (MPI) standardisation effort started [24] to define a common interface and corresponding features. The main advantages of this standard are portability and ease-of-use. The interface leaves the implementation of the message passing to the individual hardware vendors to make optimum use of the specific hardware. Some of the goals of the Message Passing Interface standard are (from [24]):

- The design of an application programming interface (API).
- To allow for efficient communication.
- To allow for implementations that can be used in a heterogeneous environment
- To allow convenient C and Fortran 77 bindings for the interface.
- To assume a reliable communication interface: The user need not cope with communication failures. Such failures are dealt with by the underlying communication system.

The use of this standard allows the creation of parallel programs which is widely portable. The programs using this communication interface my run on distributed-memory machines, networks of workstations or shared-memory machines. The actual implementation is dependant of the hardware and the vendor, but transparent to the user. Due to these advantages, MPI has become the standard for parallel computing. More information on the MPI standard can be found on the forum's web-page: `http://www-unix.mcs.anl.gov/mpi/`.

8.3.2 Data and method

First of all, the controlling module needs to know which subdomain(s) to load and where other subdomains are located. This is done at startup where the main process (process nr. 0) distributes the domains as evenly as possible and sends the results to the other processes. In the controller object all the processes now contain the list of subdomains and where they are located. Each process then loads the subdomain or subdomains as required.

At this point each subdomain needs a way of communicating with the other subdomains. This should be done without the subdomain class knowing anything about the communication, so that other programmers, who have nothing to do with the parallel programming, do not get disturbed by it. An additional class has been developed which is a base class for the subdomain class, to handle the communication between the subdomains, as shown in figure 8.2. Another advantage of this approach is that the communication class can be used for other implementations of computational classes which need to communicate. The communication class reads the node connectivity which is stored on the database and requests the location of the connected subdomains. This way the communication is completely transparent.

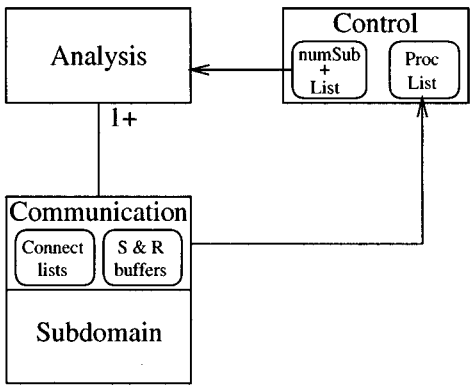


Figure 8.2: Introduction of the communication class

8.3.3 Communicating

Here the method of communicating the internal forces is described. In case the subdomain is located in the same process as the other subdomain it is communicating with, a shared memory buffer is created for sending data. The position of this buffer is exchanged with the other subdomain via the controlling object. This way one has a method of having multiple subdomains in a single process version of the program. It will also reduce communication time between subdomains which are in the same process in case of parallel computing, as reading and writing to memory is always much faster than sending messages.

For passing messages between different processes use is made of the message passing library MPI (Message Passing Interface, [11, 34]). MPI is an international standard and available on almost every platform. The basic functions are send and receive, each with a blocking and non-blocking version. A blocking send or receive is a function call which returns control back to the calling program as soon as the data is received at or sent from the other subdomain the object is communicating with. A

non-blocking function call returns immediately, but proper execution of the send or receive is not guaranteed.

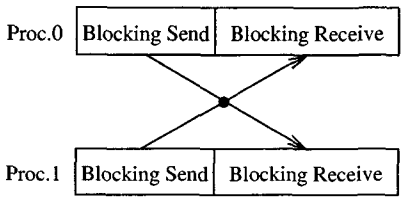


Figure 8.3: Possible deadlock due to blocking send and receive

The most direct implementation of the communication between subdomains would be that each subdomain sends its proper data and then receives and treats the incoming data. The problem with this scheme is that it would cause two or more programs to send data at the same time and to wait until the other end has received the message with the risk of deadlock, as shown in figure 8.3. Most MPI implementations provide means of avoiding this obvious possibility of deadlock, allowing this scheme to be implemented. However, some substantial additional delay in the communication is to be expected.

To avoid this risk of deadlock we can make use of non-blocking communication, e.g. a non-blocking send and a blocking receive. The problem here, however, is that the sending process has no way of knowing whether the message arrived or not. Also, a non-blocking send buffer is only sent to the other process when the other process is ready to receive. If by accident the send buffer is changed before the message is sent, the wrong information will be transferred.

To prevent both deadlock and the problem of un-received messages , the inverse scheme is used, as shown in figure 8.4. Both processes set up the receive buffers and do a non-blocking receive. Now they are ready to receive data from the other process and can send their own data by a normal send. After the send call has returned, the process knows that the proper data has been received. It now checks the receive call(s) for

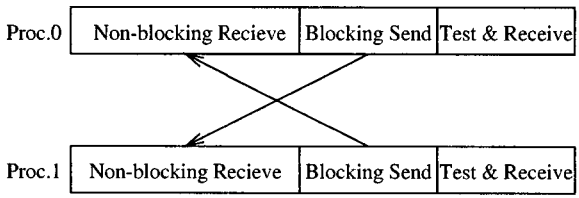


Figure 8.4: Send and receive scheme for explicit code

messages received. As soon as the proper message has been received, the process can treat the incoming data in the receive buffer. Although a receive buffer is required for each incoming message, only one send buffer is needed as messages are sent one after the other and are guaranteed to be sent so that the data in the buffer can be overwritten.

8.3.4 Synchronisation

In order to be sure that all subdomains have finished their cycle and sent and received their data, the processes have to be synchronised. The synchronisation is carried out in the control object and is combined with the synchronisation of the integration time step, which has to be the same for all subdomains. The used time step has to be minimum of all the computed critical time steps. To that effect the MPI library contains a series of synchronisation functions. The function used combines the synchronisation of the computation with the synchronisation of the time step. The function `MPI_Allreduce` collects all time steps and performs an operation on them, in this case finding the minimum, and then returns the global minimum to all processes.

8.3.5 Reading and writing to file

A neglected subject in the description of the parallel program so far is the I/O, or the reading from and writing to file, of a parallel program.

How, then, do multiple processes, which are often distributed over different machines or parts of a computer, read their information from file, and write their results to disk? And how can one prevent two processes writing to the same part of the disk at the same time? These aspects of parallel computing are seldom discussed, despite their relative importance for the execution of the program. Especially for programs which require a lot of disk access, parallel I/O can be a real bottleneck in execution time.

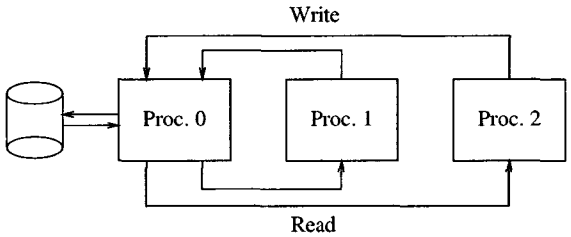


Figure 8.5: Common method of I/O in parallel program

The usual method of handling parallel I/O is by allowing only *one* process to access the disk, as shown in figure 8.5, thus preventing conflicts and allowing for all processes to read from the same file without having the data first copied to a local disk. This method, however, has some serious drawbacks. First of all the master process has to know what and how much data need to be received from each processor. This makes the I/O rigid, so that implementing flexibility is difficult. The other disadvantage is that while the master process is receiving and writing from or reading and sending to other processes no computations can be executed by this process. This creates imbalance in the parallel computation. Even by running the I/O in a different thread (difficult to implement as MPI is not thread safe) it still occupies the CPU and thereby prevents the computing thread from running.

As mentioned before, the B2000 finite element system uses the data management system MemCom for its dynamic memory management and

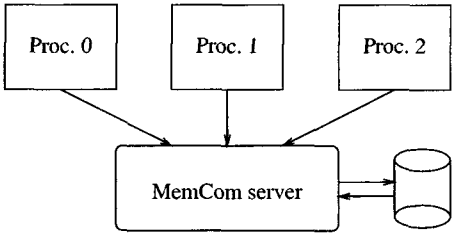


Figure 8.6: The MemCom client-server for parallel I/O

I/O to the MemCom database. This data manager is available in client-server version, alleviating the application programmer from parallel I/O problems. Each process now accesses the database through the MemCom server. The MemCom server itself is an independent program running on one of the processors of the computer or on a completely different machine somewhere on the net. This way the parallel I/O is completely transparent to the programmer and at the same time does not interfere with the balance of the parallel computation, as long as the server is not running on the same processor as one of the computing clients.

8.4 Benchmark results

The first benchmark tests with the explicit code were performed on the Swiss-T0, an eight-node ALPHA cluster with a peak performance of 500 MFLOP/s per node. This first version of the code used the blocking send-and-receive scheme as described above. File access was done locally by copying the database to each local machine.

8.4.1 Influence of the number of elements

For the benchmark simulations a simple model is used which can be easily split up in equal blocks, thereby creating well balanced computations.

Figure 8.7: Test model for benchmarks. Decomposition in 4 subdomains

The model is a beam bending problem with the beam clamped on one end and with a distrubuted load on the other end. The beam is made of volume elements and is shown in Figure 8.7. As the input file of B2000 describing a finite element model can be parametrised, this allows us to quickly modify the parameters, like the number of elements, in order to perform simulations of a different size. The number of elements of the beam was increased from 864 to 11200 elements in 5 steps and simulations were run for each model on 1, 2 and 4 processors. The results of these tests are summarised in Figure 8.8 and Table 8.2.

nr. of elements:	864	2240	5040	6400	8600	11200
S_2:	1.57	2.07	2.34	2.29	2.29	1.98
S_4:	2.23	3.40	4.06	4.11	4.23	3.62

Table 8.2: Speedup results for different problem size

These results are, although applied to relatively small models, very interesting. We see that as the size of the problems grows, the parallel performance improves. The speedup obtained is in some cases even larger than the ideal speedup. This is called super-linear speedup and can be explained by the fact that as the problem is cut into smaller pieces, more data can be stored in the cache, the small but very fast memory

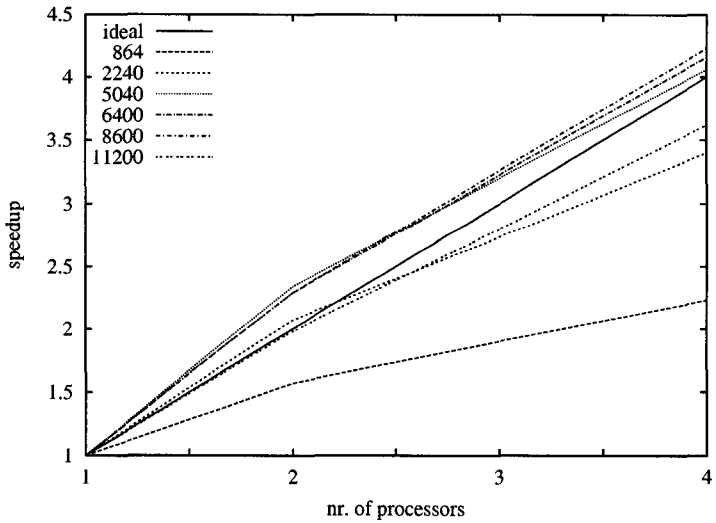


Figure 8.8: Speedup for different problem size

close to the processor. Access to the data is now substantially quicker than when the data has to be fetched from normal memory, which is a few times slower than the cache. When cutting up a problem into smaller pieces, at a certain point all the data fits into the cache, causing the super-linear speedup. As the problem size continues to grow, at a certain point even the reduced data set does no longer fit in the cache, thereby reducing the advantage. That is why the maximum speedup on two processors is obtained with 5040 elements, while the maximum speedup on 4 processors is obtained with a problem size of 8600 elements, almost twice the size of the other extreme.

8.4.2 Large models

In order to validate the performance of large models of industrial size, a model consisting of 64000 elements has been generated. The input file of this set of tests can be found in Appendix D. This model was meant to be a benchmark case of the Swiss-T1, which has a maximum number of 64 processors. However, as the use of MPI over fast Ethernet was limited to 48 processors, tests have been performed up to 32 processors only.

The serial time T_s per computing cycle (wall time) was measured to be:

$$T_s = 0.4322\,\mathrm{s} \tag{8.16}$$

and the computation to communication ratio r_p using Fast Ethernet is:

$$Tcomm_p = 2\left(\frac{0.022}{10} + 0.0005\right) = 0.0054\,\mathrm{s} \tag{8.17}$$

$$r_p = \frac{T_s}{Tcomm_p} = \frac{0.4322}{0.0054} = 80.0 \tag{8.18}$$

so that we have a theoretical speedup limit of 80 using MPICH. Using the specialised network, T-Net, which has a claimed latency of $20\mu s$ and a bandwidth of 50 MB/s we get the following communication time $Tcomm_p$ and ratio r_p:

$$Tcomm_p = 2\left(\frac{0.022}{50} + 0.00002\right) = 0.00092\,\mathrm{s} \tag{8.19}$$

$$r_p = \frac{T_s}{Tcomm_p} = \frac{0.4322}{0.0054} = 470.9 \tag{8.20}$$

We see that, mainly due to the significant deduction in communication latency the theoretical speedup is now 470!

Using Fast Ethernet

Based on the performance analysis of section 8.2 the theoretical speedup has been computed for 2, 4, 8, 16, 32 and 64 processors. These theoretical values could be compared to measurements up to 32 processors. The results are shown in Table 8.3. This shows the close relation between the theoretical and actual speedups. It also demonstrates the enormous reduction in computing time possible when using parallel computers, with a theoretical speedup of 35 using 64 processors.

P	S_p measured	S_p theory	P	S_p measured	S_p theory
2	1.92	1.95	16	12.1	13.3
4	3.64	3.81	32	19.6	22.8
8	6.84	7.27	64	-	35.6

Table 8.3: Measured and computed speedup numbers for large problem using Fast Ethernet

Using T-Net

The same performance analysis as above has been performed for the fast network available on the Swiss-T1, T-Net. Measurements have been performed up to 8 processors due to limited availability of the machine and its fast network. The results are shown in Table 8.4. It clearly demonstrates the great advantage of the low-latency communication, resulting even in a super-linear speedup when using 8 processors. The close relation between the theoretical and actual performance for both the Ethernet and T-Net network gives us confidence about the theoretical results. It demonstrates that the explicit finite element program as developed and described in this thesis is *very well* suited for parallel computing, particularly when using low-latency networks. Using 64 processors we can obtain a speedup of approximately 55 for well balanced problems.

P	S_p measured	S_p theory	P	S_p measured	S_p theory
2	1.98	1.99	16	-	15.5
4	4.04	3.97	32	-	30.0
8	8.53	7.87	64	-	56.3

Table 8.4: Measured and computed speedup numbers for large problem using T-Net

Speedup graphs on Swiss-T1

Finally we will show here the performance results on the Swiss-T1 Alpha Cluster in two graphs. Figure 8.9 shows the theoretical speedup up to 96 processors. It not only shows the less speedup obtained by using a standard network instead of an optimised low-latency network for this type of computations, it also gives a very graphical impression of the advantage of parallel computing. Figure 8.10 then demonstrates that the actual performance confirms the theoretical analysis.

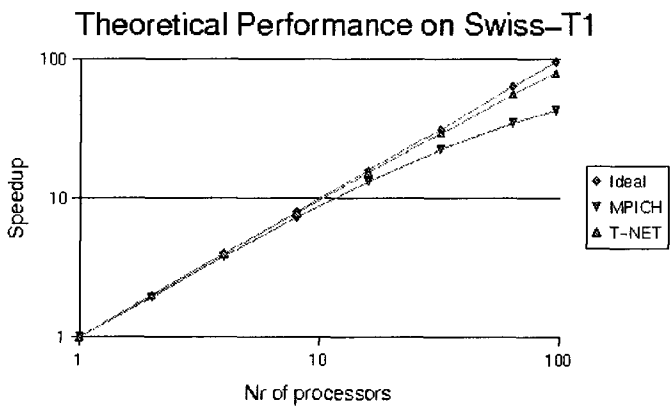


Figure 8.9: Theoretical speedup of benchmark problem on Swiss-T1

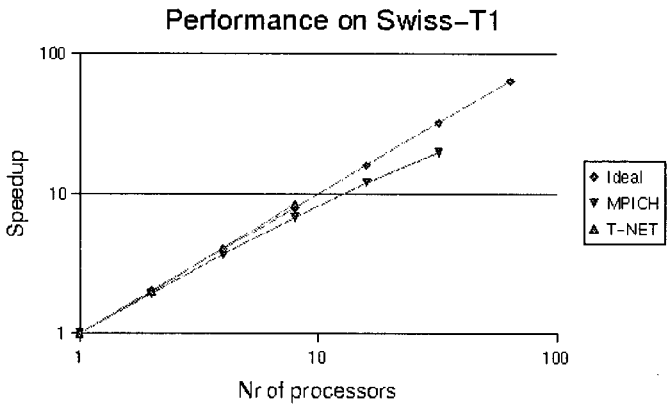


Figure 8.10: Measured speedup of benchmark problem on Swiss-T1

8.5 Conclusions

In this chapter the implementation of parallel computation of the explicit finite element method and its performance have been described. Although parallel explicit finite element programs have existed for a long time, it is shown here that the implementation of the parallel computation by means of mesh decomposition is greatly facilitated by the object-oriented structure of the finite element code. The existence of the subdomain class and the locality of data makes introduction of multiple subdomains straightforward in that you simply create multiple subdomain objects. Thanks to the decoupled equations of motion, the amount of communication between subdomains is kept to a minimum. Contrary to the parallel implementation of existing Fortran codes, the object-oriented code developed in this thesis allows for parallelism and communication to be implemented in a completely transparent way. Therefore, parallelism and communication will not be visible to most developers of the

code, ensuring easy development and reducing the risk of errors.

This chapter has also shown the performance and possible gains of parallel computing in demonstrating that the explicit finite element code is very well suited for this type of high-performance computing, allowing for the simulation of large and complex problems. The theoretical analysis of the performance of the code for a given problem shows to be in good agreement with the actual speedup of the program.

9

Applications

In this chapter some applications of the explicit finite element method are shown. These applications involve new developments, some of which make use of the parallelism in the code. The applications described here are structural optimisation, impact simulations and mode-jumping computations.

The first application is the optimisation of material parameters by simulating the material specimen test. Finding the material parameters of complicated material models is often a daunting and time consuming task. Combining the explicit finite element program with a structural optimisation tool should facilitate the task of finding the internal material parameters. This idea has been jointly developed and implemented with P. Arendsen of the Dutch aerospace laboratory NLR in the Netherlands. Currently, no working optimisation tools for this kind of problems are available.

The second application is the simulation of impact on structures. Although well established for metal structures, the simulation of impact for composite structures has up till now been more complicated due to the absence of sufficiently accurate composite damage models. Using the UD composite damage model described in chapter 6, impact simulations

have been performed which validate the usefulness of this model.

Finally, the third application is the use of explicit finite element methods to simulate the dynamic behaviour of a curved shell after reaching its limit load. The study of this behaviour has become of interest over the last few years and important contributions to the subject have been made by Riks and Rankin [30, 31]. As the non-linear quasi-static analysis up to the limit point is done with implicit codes, the time integration is traditionally also performed using implicit codes with implicit time integration schemes, like Park's [25]. However, due to its very dynamic non-linear behaviour, the time step required for a convergence of the solution has to be very small. The question therefore arises, why an explicit method is not used for the dynamic simulation. Theoretically there is no reason why the explicit method could not be used and this question was already discussed by the author with E. Riks, G. Rebel and J. Remmers at the Delft University of Technology in 1996 and 1997. In section 9.3 a first attempt is made to couple the continuation routine of the implicit finite element code with the explicit finite element program developed in this thesis.

9.1 Optimisation of material parameters

In continuum mechanics the theory of elasticity is well established. But with increasing demands on the behaviour of structures, the plastic and fracture response of the material need to be studied. This requires very complex material models with additional internal parameters, especially in the case of anisotropic materials, as shown in chapter 6. Obtaining these internal material parameters is usually a complicated task. Based on specimen tests stress-strain curves are obtained by interpreting the test data. Then the material parameters are chosen in such a way as to approach the stress-strain curve as closely as possible. This is a complicated and intensive task.

Together with the NLR, the University of Oxford and the University of Linköping, a more direct approach has been proposed. By directly

simulating the material test and integrating the simulation tool in a structural optimisation tool, the internal material parameters can be obtained directly.

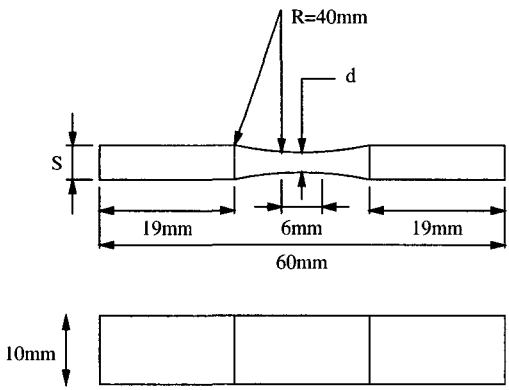


Figure 9.1: Dimensions of test specimen

In order to evaluate this new approach, material tests performed by the University of Oxford on unidirectional carbon fibre material have been simulated using the material model from the University of Linköping, described in section 6.1. The explicit code B2ETA has been integrated in the optimisation module of B2000 in order to perform the optimisation. This module, B2OPT, has been developed at the NLR, which also did the integration with the explicit code. Simulations have been performed jointly with the NLR.

9.1.1 Specimen

The material test specimen is a waisted strip measuring 60 mm long and 10 mm wide. On the edges the laminate consists of 18 layers of carbon fibre which have been waisted down to 8 layers in the middle. The dimensions of this specimen are shown in figure 9.1.

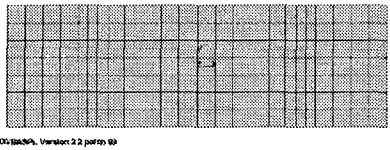


Figure 9.2: Initial mesh for specimen test simulation

The initial mesh is shown in figure 9.2. The test data available are the velocities on both side on the specimen, the reaction force on the (left) end of the specimen and strain measured in the middle of the specimen. Using the velocities on both sides of the specimen, the reaction force was taken as the reference to which the optimisation has been performed. Figure 9.3 shows the measured reaction force of the test with the fibres in the 90^0 direction. A series of points on the curve are used as reference values for the optimisation.

The material model used in this case is the Linköping model described in section 6.1. This model for thin plates (2D) not only requires the usual elasticity parameters (E_1, E_2, ν_{12} and G_{12}), but 16 additional internal parameters, which brings the total to 20 material parameters per material type for this model. (Compared to 4 for normal anisotropic plate materials.)

9.1.2 Optimisation method

The Dutch aerospace laboratory NLR has a large experience in the optimisation of structures using static analysis. An optimisation program has been developed around the finite element package B2000. As the explicit program is a module of this environment, it uses the same data structure as the implicit code. This allowed us to integrate the explicit code in the existing optimisation program. The main difference between the implicit and the explicit analysis is the fact that in the explicit code no perturbation matrix can be constructed to guide the optimisation in the proper

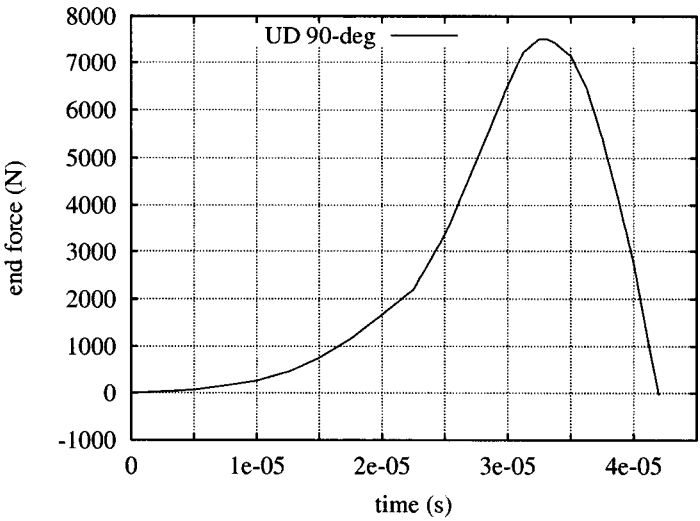


Figure 9.3: Measured reaction force on specimen

direction. Therefore, the parameters to be optimised are perturbed and for each perturbed optimisation parameter a new simulation is ran in order to obtain an optimisation direction. The whole procedure is shown in figure 9.4. In order to smoothen and improve the optimisation, a second order scheme is used for the direction search. The integration of the explicit code with the optimisation program, as well as the actual optimisation itself, was done by P. Arendsen. Here, once again, the object-oriented programming approach proved to be very useful. Instead of introducing `ifdef` statements in the code for the optimisation version and the standard version, function overloading was used within the control class.

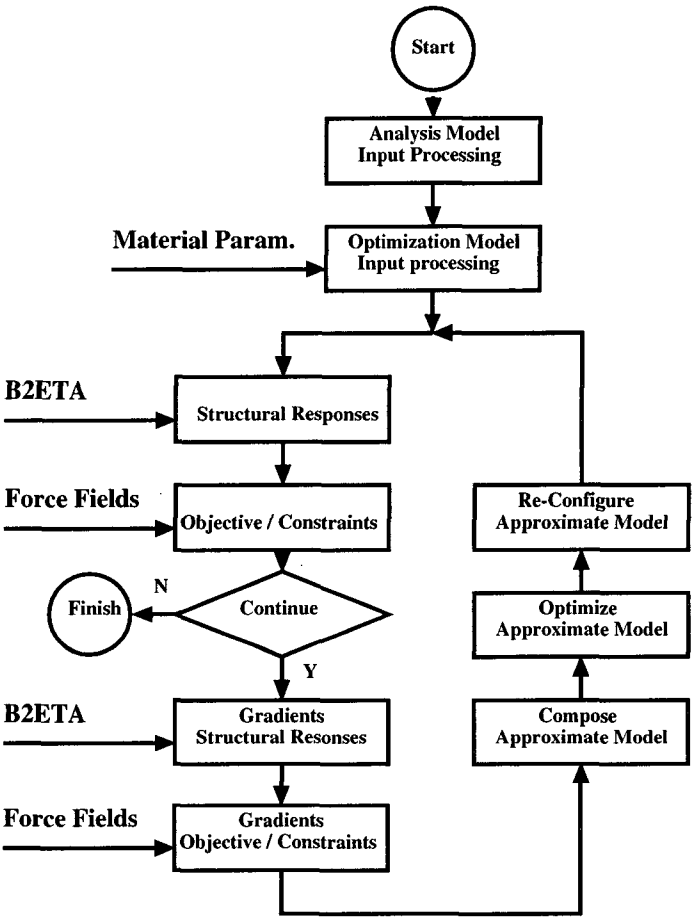


Figure 9.4: Schematic of the optimisation procedure

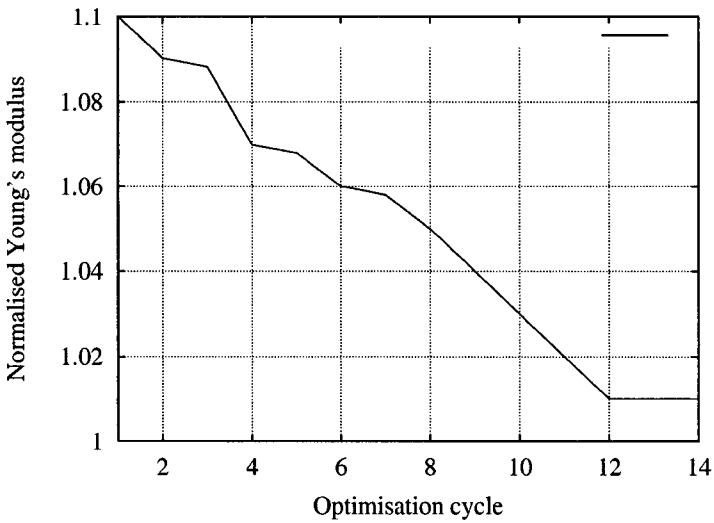


Figure 9.5: Normalised value of E_1 during optimisation run

9.1.3 Results

In order to demonstrate the concept to be working, a simulation was run with the initial estimated material parameters. The results from this simulation were used as the reference solution. The modulus of elasticity in the fibre direction (E_1) was changed to a value 10% larger than the actual value. Then, the optimisation procedure was run to find the correct value of the Young's modulus in the fibre direction. The maximum change of the value per cycle was limited to 1%. The results of the optimisation are shown in figure 9.5 and figure 9.6. The first figure shows the normalised value of the modulus of elasticity, the second figure shows the error of the simulation over the entire time interval. The error of the simulation is

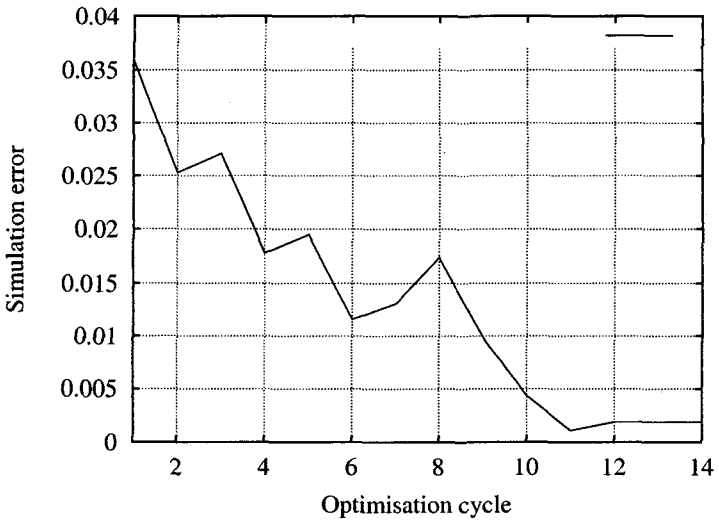


Figure 9.6: Simulation error during optimisation run

given by:

$$\sum_{i=0}^{i=N} \left(\frac{(X_{ref} - X_{sim})^n}{X_{ref}^m} \right)_i \tag{9.1}$$

where N is the number of points on the force-time curve, X_{ref} the reference or measured value, X_{sim} the value of the simulation and n and m weight exponents which are to be chosen. Usually $n = m = 2$.

Figure 9.5 shows that the modulus of elasticity descents slowly to the correct value. Figure 9.6, however, shows the difficulties encountered during the optimisation: The error with respect to the reference curve does not show a descending trend with each optimisation cycle. This is because of the dynamic effects in the simulation and the occurrence of material fracture, increasing the non-linearity of the response. Thanks

to the use of the second order polynomial used for the optimisation direction, the parameter to be optimised continues to approach the correct value with each cycle.

The results with the optimisation described above demonstrates the program to work properly, despite the large non-linear effects. This gives us confidence in the success of the optimisation for the material parameters. Unfortunately, final results are not yet available, because the analysis is still going on. The reason for this is that the modelling of the specimen tests turned out to be more complicated than initially thought. First of all, comparing the initial response of the simulation with the response measured in the test showed a shift in the force-time curve. The reason for this shift was found to be in the definition of the boundary conditions. The prescribed velocity field was initially applied to the specimen at the end of the impact bar, while in reality this was the velocity at the beginning of the clamp, i.e. at the end of the specimen in the clamp. Therefore, the length of the specimen model needed to be extended. Also the mesh was refined to improve the accuracy of the solution, particularly with respect to the fracture of the elements. The resulting mesh is shown in figure 9.7.

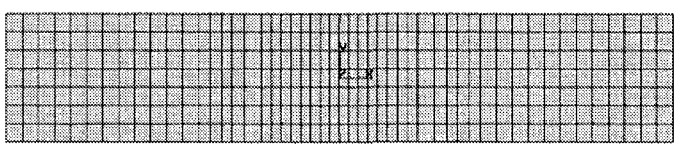


Figure 9.7: Adapted model for specimen test simulation

The actual test specimen with the fibre direction in the length of the specimen was waisted not only in the thickness, but also in the height of the specimen to reduce the maximum stress in this test. This required an

additional modification for this particular test case. The mesh of the test specimen with the fibres in the 0-degree direction is shown in 9.8.

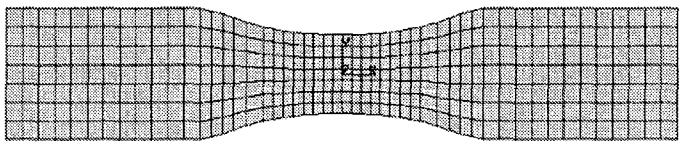


Figure 9.8: Model for 0-degree test specimen simulation

9.1.4 Conclusions

In this section, the coupling between an optimisation program and the explicit finite element program has been described. The resulting program has been applied to the optimisation of material parameters by simulating the actual material tests. The proof-of-concept simulation of a perturbed modulus of elasticity with respect to a reference simulation demonstrates that the method is working. It also shows the complexity of the optimisation due to the highly non-linear response of the specimen. The first simulation runs showed that the finite element mesh needed modification in order to model the actual tests correctly.

The method described in this section shows great potential for the derivation of internal parameters for complex material models. Once a material test has been performed and a corresponding finite element model with the proper boundary conditions has been made, the program can be used to find the material parameters of any implemented material model. Once the material parameters of a specific material are known the model can be used in structural simulations.

Unfortunately, final conclusions can not be drawn yet, as the work is still going on. In order to get all the material parameters (20!) based

on only 2 material specimen tests, multi-model simulations have to be performed. This means that the optimisation direction of the parameters is based on the response of all simulations at the same time. However, based on the first simulations and tests, we are confident that the final results are positive. Although the procedure can be quite computationally intensive, the optimisation method itself is very well suited for parallelisation (multiple simulations at the same time), which can greatly reduce the total optimisation time.

9.2 Impact simulations

Impact of objects is a very important design criterion for structures in aerospace, automotive and civil engineering. This type of problem has always been very difficult to study. With the modern level of computing power and finite element programs, the study into the behaviour of structures under impact up to the point of failure becomes possible. However, these kind of simulations are inherently very intensive in terms of computing, and can be solved within reasonable time only by parallel computing. Here, we will describe the simulation of a gelatine cylinder impacting a composite plate. This simulation corresponds to structural tests performed at the University of Oxford and makes use of the UD material model described in section 6.1 and the gelatine model described in section 5.2.

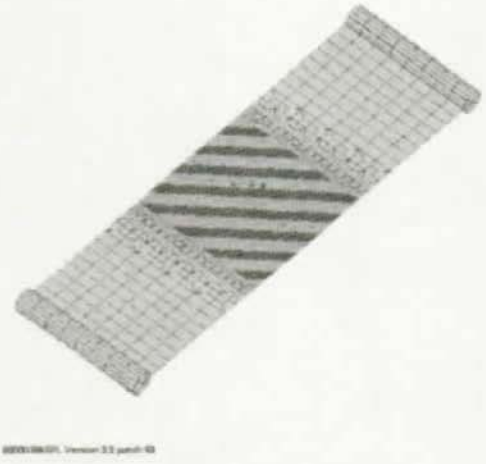


Figure 9.9: Structure test simulation

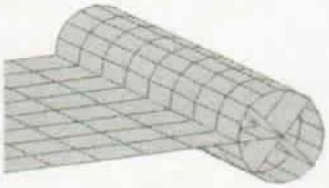


Figure 9.10: Modelling of structure clamp

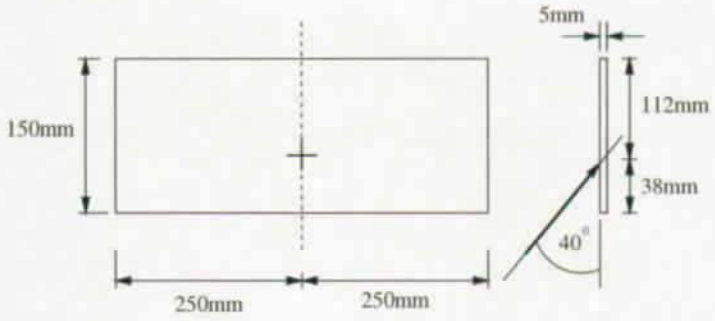


Figure 9.11: Structure test simulation

9.2.1 Structural model

The model consists of a composite plate clamped on both sides by a cylinder as shown in detail in figure 9.10. The clamps of the structure had to be modelled as well, because for low velocity impact the plate was damaged at the edges. The actual tests had a gelatine cylinder impacting the plate at a 40 degrees angle, as shown in figure 9.11.

9.2.2 Metal impact simulation

The first simulations were not done with gelatine as an impactor, but with a small block of aluminium, as shown in figure 9.12. The block touched the structure at the same point of impact but perpendicular to the plate. With a speed of 800 m/s it was meant to deposit approximately the same amount of energy as the high-speed gelatine impact. The reason for this simulation was to test the behaviour of the damage model, and to test the capabilities of the finite element program without the additional complexities of the gelatine model.

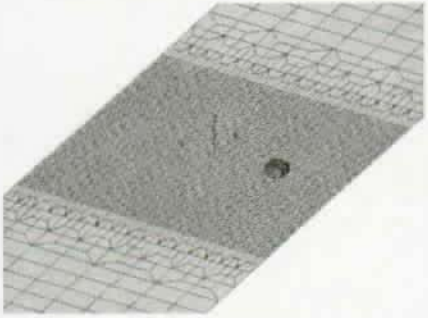


Figure 9.12: First simulation: Metal block impact

The result of the metal block impact is shown in figures 9.15 and 9.16. At the initial impact the composite plate breaks at the point of impact to form a hole through which the impacter falls. Due to dynamic and bending effects, the plate breaks into two parts somewhat later. This corresponds to the behaviour seen in similar impact tests.

9.2.3 Gelatine impact simulations

The gelatine simulations use the gelatine model described in section 5.2. The structure with gelatine model is shown in figure 9.13. They correspond to actual tests performed at the University of Oxford [13]. The

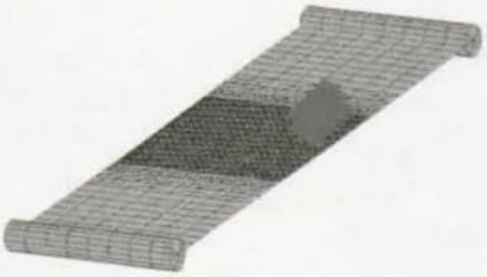


Figure 9.13: Gelatine impact simulation: Initial situation.

problem with the impact simulations was the proper behaviour of the gelatine. As the parameters of the inter-particle law were not known in advance, they had to be estimated based on the impact simulations on the composite structure. This was done by simulating the low-speed impact (230 m/s), as shown in figure 9.14, and comparing the results with the actual test data. The parameters obtained from this simulation were then used in the other simulations with higher speeds of impact. However, it turned out, that the behaviour of the gelatine is *very rate-dependent*. Several iterations were needed to approach the actual behaviour of the gelatine. The shape of the gelatine impactor in the simulations remains the original cylinder, while the actual impactor changes shape with varying impact speed. This is already shown in figure 5.8. All these effects had a large influence on the simulation results. The final damage of the simulations with an impact speed of 230 m/s, 390 m/s and 480 m/s is shown in figures 9.17, 9.19 and 9.21. The simulations can be compared to the results of the actual tests, which are shown in figures 9.18, 9.20 and 9.22.

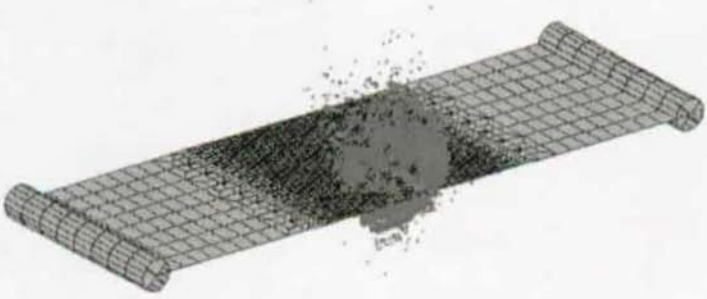


Figure 9.14: Gelatine impact simulation: Gelatine behaviour at 230 m/s.

Impact speed of 230 m/s

The low impact speed of 230 m/s caused only minor damage and fracture of the structure at the edges of the plate near the clamps, as shown in figure 9.18. This is the reason the boundary conditions needed to be modelled as well. Although the superficial damage is small, C-scans will probably indicate substantial internal damage. The simulation shown in figure 9.17 see the creation of a hole in the middle of the plate, as well as considerable damage of the elements near the clamps. Because of the parameters of the inter-particle law used in the simulation, a too stiff response of the gelatine concentrated too much mass at the centre of the plate, causing the damage in the middle of the plate. A too coarse mesh near the clamps caused the substantial damage at the edges of the plate. When in an element the maximum allowable damage has been reached, the element is taken out of the computation. Due to the large element size near the clamps, this results in a sudden substantial loss of mass and material able to absorb the energy of the impactor. With smaller elements this loss of structural integrity this effect would be much more gradual,

thus giving better results.

Impact speed of 390 m/s

The medium impact speed of 390 m/s caused substantial damage in the middle of the plane, as shown in figure 9.20. The impacting gelatine caused serious damage to the plate with a large loss of mass in the impacted region. The simulation, figure 9.19, shows a good correlation with the actual damage. This demonstrates that the gelatine model is capable of depositing mass in a proper way, and that the program is capable of simulating impact tests with a reasonable degree of accuracy.

Impact speed of 480 m/s

The high impact speed of 480 m/s caused relatively small damage, creating only a hole in the middle of the plate, as shown in figure 9.22. The simulation results, however, show a much larger damage area, as can be seen in figure 9.21. This could be because of a too soft response of the gelatine model. But more likely the small damage in the actual test is due to the change in shape of the impactor. The shape of the gelatine cylinder changes to a cone, as shown in figure 5.8c. This gives a much smaller area of impact, resulting in a more concentrated load on the plate. Changing the shape of the impactor in the numerical simulations could improve the results.

9.2.4 Conclusions

The simulation of the impact of a composite plate is an example of a modern industrial application. This type of simulation is well established for metal structures with solid impactors. Gelatine is also used in simulations by using fluid-like elements, but has the numerical problems of large deforming and imploding elements, and the loss of connectivity. The gelatine model used here does not possesses these problems, but finding the

proper values for the parameters of the inter-particle law is more complicated. The use of composite instead of metal for the structure is an additional complication. The simulations show potential promising results, but require continued research, especially into the behaviour of the gelatine.

The impact simulations greatly benefit from the possibility of parallel computation. The simulations described above take up to 8 hours to compute on a single processor[1] due to the run-time reduction of the time step. This time step needs to be scaled down because of failure of layers in the composite plate. This failure of the layer results in a reduced effective thickness and an increase in the eigenvalue of the shell element. Not scaling down the time step results in divergence and instability of the solution. With computation times this long, parallel processing is necessary to solve these problems in a reasonable amount of time.

9.2.5 Simulation figures

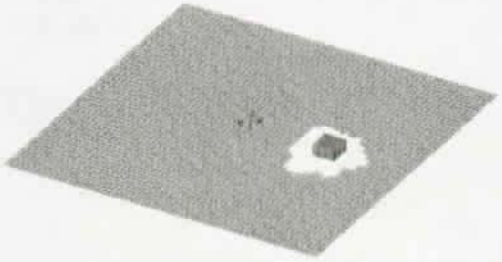


Figure 9.15: Metal block impact: initial penetration

[1] Intel Pentium-II, 333MHz

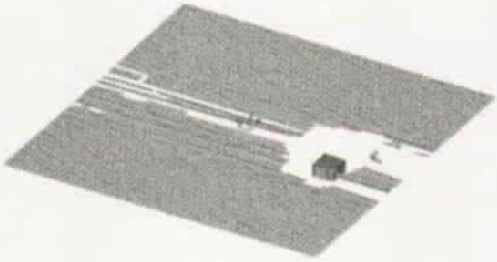


Figure 9.16: Metal block impact: final damage

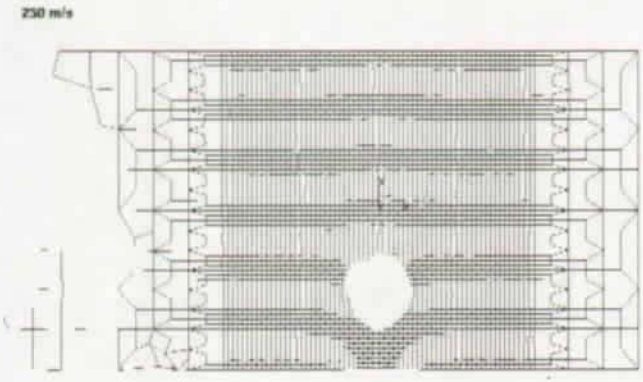


Figure 9.17: Gelatine impact: simulation at 230 m/s

Figure 9.18: Gelatine impact: test result at 350 m/s

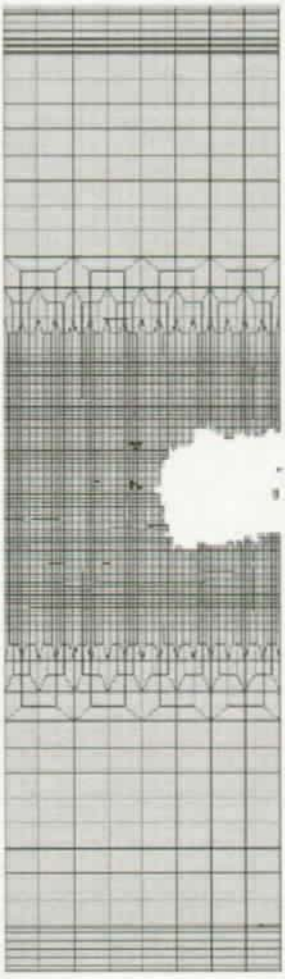


Figure 9.19: Gelatine impact: simulation at 390 m/s

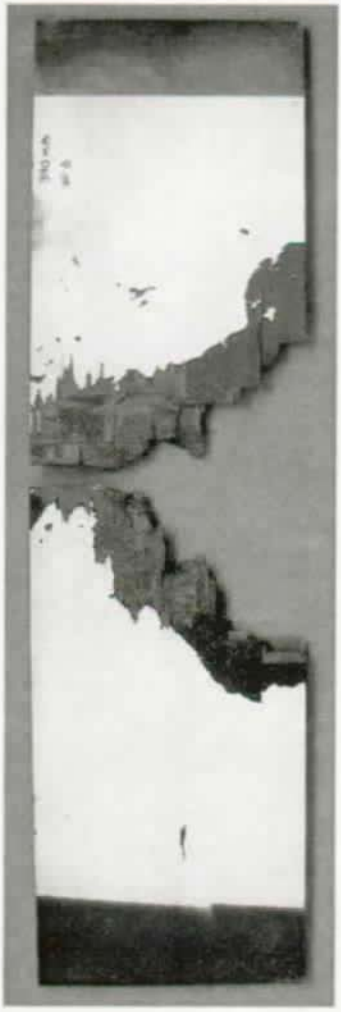


Figure 9.20: Gelatine impact: test result at 390 m/s

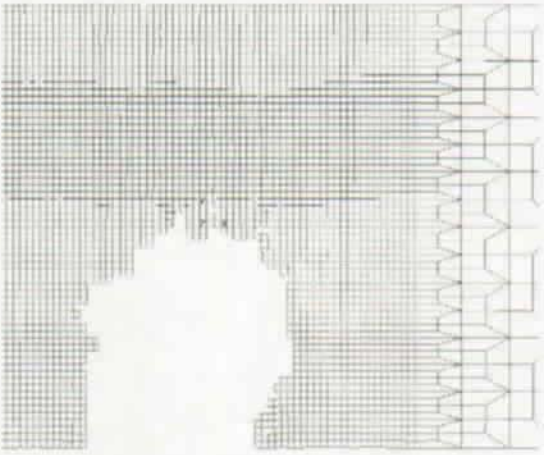


Figure 9.21: Gelatine impact: simulation at 480 m/s

Figure 9.22: Gelatine impact: test result at 480 m/s

9.3 Mode-jumping simulations

In non-linear quasi-static analysis, a structure can undergo a sudden change in its state. This phenomenon in called *mode jumping* and happens during buckling analysis when the structure jumps from one stable state to another, with a sudden change in wave numbers. This phenomenon was first mentioned by Stein [35] when describing a buckling experiment of a flat panel. The answer to this problem is sought in the stability of buckling modes. When a structure buckles its equilibrium state can become *unstable*. The structure will then move to its nearest *stable* equilibrium position.

This process is highly dynamic and therefore requires a dynamic simulation method. A lot of work in the simulation of this phenomenon is done by Riks *et al.* [30, 31]. As stability analysis is performed by implicit methods, the time integration is also performed with implicit time integration schemes, because implicit and explicit codes have always been two different worlds. These schemes, however, are inherently computationally intensive and thus slow. A possible way of speeding up such computations is by using an explicit finite element code for the dynamic simulation. This requires a conversion of the data from the implicit computation and an integration of the two codes. As implicit continuation methods and explicit time integration methods are two very different branches of the finite element tree, no reference in literature could be found. The work described in this section has been carried out in close collaboration with J. Remmers and E. Riks of the Delft University of Technology.

This section describes the implementation and results of the linking of implicit and explicit methods for mode jumping simulations. First a brief overview of stability analysis and the continuation method is given. Then the transient mode jumping process is discussed. Both subjects are discussed only briefly here, as an excellent description of the phenomena and the solution methods used can be found in [28], on which subsection 9.3.1 and 9.3.2 are based. Subsection 9.3.3 describes the linking

of the continuation module of B2000 to the explicit code followed by two examples in 9.3.4: The deformation and vibration of a beam and the buckling and post-buckling behaviour of a curved panel. Finally, subsection 9.3.6 discusses the first results and draws some conclusions as to the possibilities of implicit-explicit coupling.

9.3.1 Continuation methods

In order to study stability and other quasi-static non-linear behaviour of structures, the deformation path of the structure must be followed. One must be careful about the solution method and the convergence of the procedure, as at a certain load multiple solutions can exist, as shown in figure 9.23.

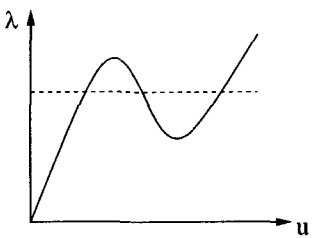


Figure 9.23: Multiple solutions of a non-linear problem

When a static equilibrium is considered, the set of equations can be written as:

$$\mathbf{f}(\mathbf{u};\lambda) = \mathbf{0} \tag{9.2}$$

where $\mathbf{f}$ is the total resulting force vector acting on the nodal degrees of freedom, $\mathbf{u}$ the vector of degrees of freedom and λ the load factor, describing the magnitude of the external forces or prescribed displacements. This is a system in $\mathbb{R}^N$, but with $N + 1$ unknowns.

To follow the non-linear solution path one can reduce the number of unknowns by selecting one unknown, the so-called path parameter and denoted η, and keep this value constant for each step. For this, one can choose either the load factor λ (incremental load method) or one of the displacements u (incremental displacement method) and increase its value for each step. After each increment of the path parameter an iteration method, like Newton-Raphson, is required to find the new equilibrium [44]. The disadvantage of both methods is shown in figure 9.23. The incremental procedure does not get the solution beyond the limit point λ^* or u^*. Decreasing the path parameter at this point would result in sliding back along the curve.

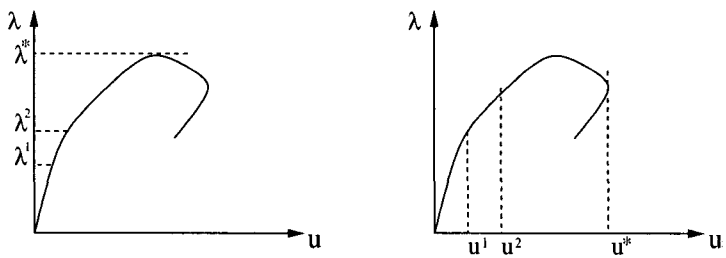


Figure 9.24: Incremental load and displacement procedure

In order to overcome this kind of problems, one can increase the number of equations by introducing a new independent equation $\tilde{f} : \mathbb{R}^{N+1} \rightarrow \mathbb{R}^1$ including the path parameter η. A new system is created with $N+1$ unknowns, now in $\mathbb{R}^{N+1}$ space:

$$\hat{\mathbf{f}} = \begin{bmatrix} \mathbf{f}(\mathbf{u};\lambda) \\ \tilde{f}(\mathbf{u};\lambda;\eta) \end{bmatrix} = \mathbf{0} \qquad \hat{\mathbf{f}} : \mathbb{R}^{N+1} \rightarrow \mathbb{R}^{N+1} \tag{9.3}$$

The simplest form of the additional equation you get by choosing the load factor as the path parameter. By choosing

$$\tilde{f} = \eta - \eta_{i-1} = \lambda - \lambda_{i-1} = 0 \tag{9.4}$$

where λ_{i-1} is the load factor of the previous step, we get the incremental load method as described above.

In order to improve the performance of the method and to make it suitable to pass over the limit points, adaptive parametrisation methods have been developed. Riks [29] derived the following additional equation:

$$\tilde{f} = \mathbf{n}^T(\mathbf{u} - \mathbf{u}_{i-1}) - \eta = 0 \tag{9.5}$$

where $\mathbf{n}$ is the direction in which the new equilibrium is sought. It is clear that the best results are to be found when the auxiliary surface, on which the iteration is performed, is perpendicular to the solution. Therefore the direction chosen is tangential to the path:

$$\mathbf{n} = \frac{d\mathbf{u}(\eta)}{d\eta} \tag{9.6}$$

This is also shown in figure 9.25. The additional equation now becomes:

$$\tilde{f} = \frac{d\mathbf{u}(\eta)}{d\eta}(\mathbf{u} - \mathbf{u}_{i-1}) - \eta \tag{9.7}$$

For each step the equation is updated and an iteration method is used to return to the equilibrium state. This whole procedure is implemented in the continuation routine of B2000, B2CONT.

9.3.2 Mode jumping phenomenon

The interest in coupling the explicit transient analysis processor with the continuation processor is to reduce the computation time on mode-jumping simulations. In this subsection the principle of this phenomenon will be described.

Sometimes during non-linear deformation of a structure, particularly thin-walled shell structures under compressive loading, a sudden loss of structural stability can occur. This loss of stability can be in the form of a

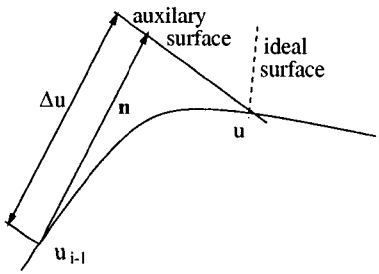


Figure 9.25: Riks' path following method

bifurcation point, where two stable solution paths cross each other, or in the form of a limit point, a local or global maximum on the deformation curve. In case of loss of stability, the structure is no longer able to carry an increasing load. At this point, $(\mathbf{u}_{lim}, \lambda_{lim})$, the path can only be followed physically by reducing the load. However, with a decreasing load factor, the path will be followed backwards.

Some structures, however, regain their stability after a new equilibrium path has been found, and can continue to carry an increasing load. An example of such a structure is the upper wing skin of an aircraft. The transition of the first limit point to the new equilibrium situation with the same load factor, is a highly dynamical process and needs to be calculated as such. Due to the high velocities and the non-linearities involved, transient solution methods need a very small time step to follow the behaviour and to arrive at the proper stable path. This kind of simulation is very well suited for explicit analysis. However, as implicit and explicit finite element codes are often two very different programs, coupling of a continuation method with an explicit time integrator is not very common and no reference in literature could be found.

9.3.3 Numerical implementation

The implementation of the coupling between the implicit continuation module B2CONT and the explicit time integrator is done by a new program which converts the data into B2ETA readable format. This program, B2I2E, assumes that the element type used in the continuation module is a 4 node shell element as developed by G. Rebel [27]. This element is a shell element with 6 degrees of freedom per node, where the last 3 degrees of freedom describe the rotation of the node. The shell element in B2ETA has 5 degrees of freedom per node. The rotations are described by the position of the unit outward normal on the nodal point, called a director. This translates into the code as 3+3 degrees per nodal point. For the explicit code to read the displacement vector, the real rotations must be converted to the proper position of the director. Stored on the database are the rotations of the node. The new rotational position of a node $\bar{\phi}$ can be written as:

$$\bar{\phi}_i = \phi_i + \Delta\phi_i \qquad i = 1, 2, 3 \tag{9.8}$$

where ϕ_i is the original angle and $\Delta\phi_i$ the increment found on the database. For the unit outward normal this means:

$$\phi_i = \arccos(\mathbf{e}_i \cdot \mathbf{n}) = \arccos(n_i) \tag{9.9}$$

Thus we get for the new direction of the nodal director:

$$\bar{n}_i = \cos(\bar{\phi}_i) \tag{9.10}$$

after which the director has to be normalised. This director replaces the rotational degrees of freedom in the displacement vector.

The problem with this approach is that information on the rotational position of the nodes is lost. This will make restart from the explicit mode jumping simulation to the continuation method, needed in order to continue following the new branch, more complicated. However, as the first step of the restart is done to get back to the static equilibrium, and

the mode-jumping simulation is mainly meant to get to the new branch as closely as possible, this should, at least for the less complicated cases, be of lesser concern. As our interest here is limited to the investigation of the possibility of using the explicit finite element method for the mode jumping simulations, the second restart problem will not be further dealt with.

In addition to the conversion of the displacements, the velocity vector must be created and stored on the database. This is straightforward, because the initial velocities of the jump after the quasi-static analysis are zero.

The last information required to make a restart with the explicit code is the element stresses. The Rebel elements do not, in their current form, store stresses, but only internal forces. Therefore, these stresses have to be computed. However, the elements do store for each step the change of the local element frames ξ and η in the integration points. The Rebel elements use four-point integration over the surface, while the explicit shell element uses only one-point integration. The change of the element frames in the centre of the element becomes:

$$\xi_i = \frac{1}{4}\sum_{j=1}^{4} \xi_i^j \qquad i = 1,3 \tag{9.11}$$

$$\eta_i = \frac{1}{4}\sum_{j=1}^{4} \eta_i^j \qquad i = 1,3 \tag{9.12}$$

where ξ_i and η_i are the new element frames in the mid-point of the element, and ξ_i^j and η_i^j the element frames in the 2x2 Gauss points of the Rebel element. The engineering strain in the local element frame can

now be computed as:

$$\epsilon_x = \sqrt{\xi_i \xi_i} - \sqrt{\hat{\xi}_i \hat{\xi}_i} \tag{9.13}$$

$$\epsilon_y = \sqrt{\eta_i \eta_i} - \sqrt{\hat{\eta}_i \hat{\eta}_i} \tag{9.14}$$

$$\gamma_{xy} = \tfrac{1}{2}(\sqrt{\xi_i \eta_i} - \sqrt{\hat{\xi}_i \hat{\eta}_i}) \tag{9.15}$$

Assuming linear elastic behaviour of an isotropic material, we can compute the membrane stresses of the Cauchy stress:

$$\sigma_x = \frac{E}{1-\nu^2}(\epsilon_x + \nu \epsilon_y) \tag{9.16}$$

$$\sigma_y = \frac{E}{1-\nu^2}(\epsilon_y + \nu \epsilon_x) \tag{9.17}$$

$$\tau_{xy} = 2G\gamma_{xy} \tag{9.18}$$

We further assume that for bending the most important factor is the stiffness and not the bending stress, thus keeping the bending stresses to zero initially. In later simulations the bending stresses have been computed as well. Together with a copy of the element connectivity list, the data set containing the integration time at this cycle ($t = 0.0$) is stored on the database thus providing the necessary information to make a restart with B2ETA for the simulation.

The use of the conversion program as well as the proper way of controlling the simulations are described in appendix C.2.

9.3.4 Numerical examples

Clamped beam

The first test case of the coupling between the continuation processor and the explicit transient analysis processor is a simple beam made of 10 shell elements and clamped on one end, as shown in figure 9.26. On the free end the beam is loaded in tension. With the continuation processor the

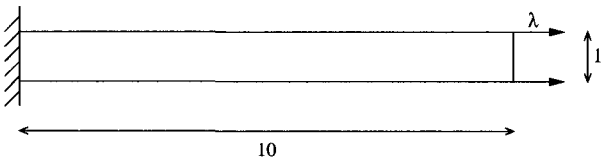


Figure 9.26: Test case: clamped beam

structure is loaded to a load factor $\lambda = 5$. From there on the structure is released and the dynamic behaviour is simulated. The simulation was run with both the implicit and the explicit transient analysis processor to compare results and computational performance.

Figure 9.27 shows the tip displacement of the beam during the vibration. Both the implicit analysis with B2TRANS and the explicit analysis with B2ETA show the same global behaviour with the same frequency. However, the explicit simulation shows a slightly longer period of the vibration. This difference has been noticed before in [28]. Reducing the time step for the implicit method will probably reduce the difference between the two methods. For the free vibration, we now also see a different numerical effect of the two time integration methods: the implicit method (Park's multi-step method) is slightly more damped than the explicit method. Both the implicit and the explicit time integration methods show the proper damping behaviour: for a critical damping factor of $\xi = 0.2$, the amplitude of the vibration is reduced by 50% in approximately half a cycle.

The computational times[2] are shown in Table 9.1. We see here a remarkable difference in performance between the two methods. Although the explicit method needs to perform twelve times more iterations, it is almost forty times faster than the implicit method. Although this is not necessarily predictive of more complicated simulations, our clamped beam example shows the possibilities of explicit analysis for this type of

[2]Run on an IBM laptop, Intel Pentium II-266 MHz.

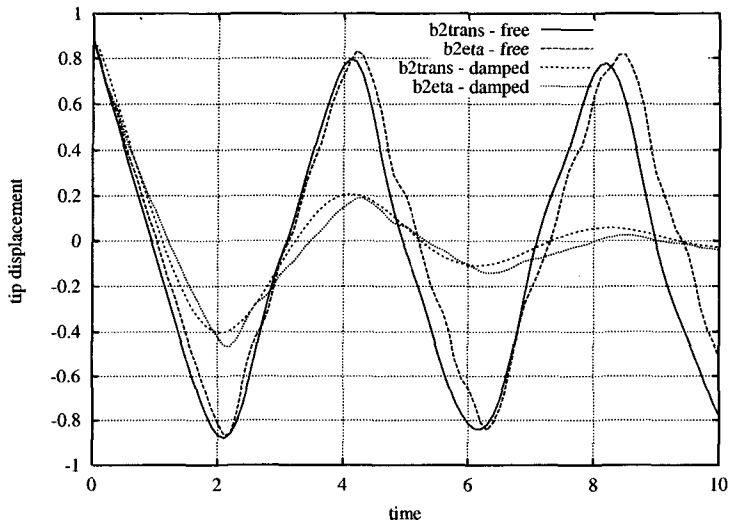


Figure 9.27: Tip displacement of beam: free and damped vibrations

computation.

The Verolme panel

The Verolme panel is a curved shell structure as shown in figure 9.28. The edges II and IV are simply supported, while the edges I and IV are clamped. The prescribed loading is the displacement δ of edge I. This structure has been extensively analysed previously [37]. With the continuation method the path beyond the limit point is computed. The dynamic simulation is started just before the limit point while still in the pre-buckling stage as shown in figure 9.29.a. The panel is then compressed beyond this limit point. Using the implicit time integrator the mode-jump has been analysed in [28]. The result of this simulation demonstrates that the next stable position is where the panel has three half waves in the

Module:	CPU time (s)	Nr. of steps
B2TRANS	62.5	100
B2ETA	1.6	1372

Table 9.1: Computational performance of B2TRANS and B2ETA

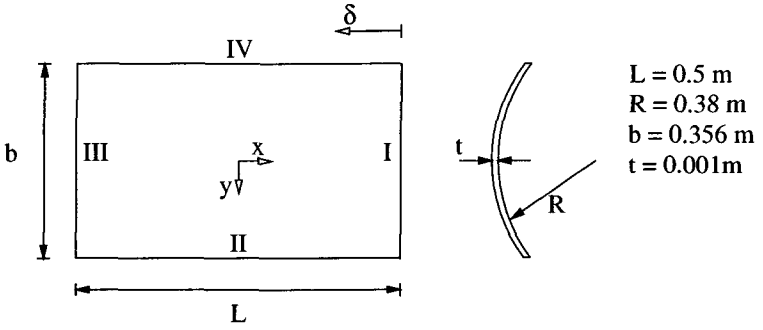
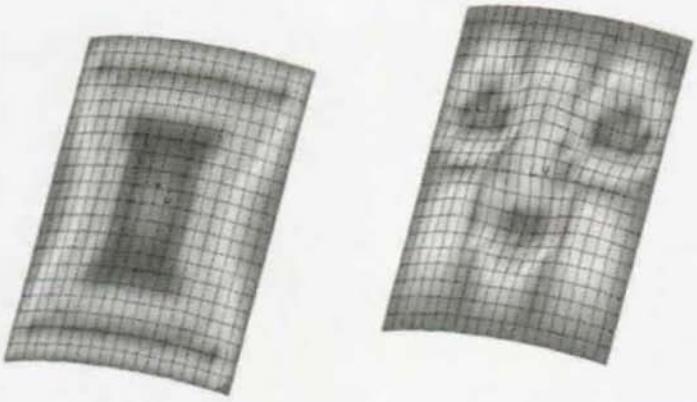


Figure 9.28: Verolme panel

upper part, and five half waves in the lower part of the panel, as shown in figure 9.29.b.

The first results with the explicit time integration gave results which showed an irregular, somewhat asymmetric deformation pattern, as shown in figure 9.30. After closely studying the results, it was found that the error occurred already in the initial phase of the simulation. By the conversion of the strains from the implicit to the explicit element a situation is created which turns out *not* to be in equilibrium. The reason for this has to be sought in the presence of zero-energy or so-called hourglass modes as described on page 18. The stresses computed in relation to the strains of the element do not take into account the hourglass deformation. The subsequently computed internal forces do not contain the contribution of the hourglass forces, thereby creating a situation which is not in static equilibrium. The simulations therefore do not result in the proper stable



(a) Pre-buckling state (b) Mode jump, t=7.45 ms

Figure 9.29: Verolme panel: Implicit analysis

mode, but converge to some irregular deformation.

In order to perform the mode jump analysis with an explicit finite element code properly, one of the two following solutions needs to be applied:

1. Compute the corresponding hourglass forces when converting from an element with 2x2 Gauss integration to an element with one-point integration, in such a way that the resulting internal forces give a system in static equilibrium.

2. Use an element in the explicit code with 2x2 Gauss integration, preferably using the same theory as for the element in the implicit code.

Computing the hourglass forces is a complicated task, due to the fact that the hourglass control mechanism used in the Belytschko-type shell element is based on a rate expression, similar to the one described in section 5.1.

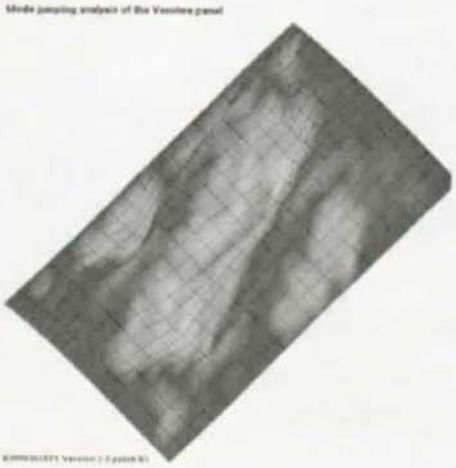


Figure 9.30: Initial mode-jumping results with B2ETA : asymmetric pattern

By far the best solution is the introduction of a shell element with 2x2 Gauss integration. This will create a direct relationship between the element in the implicit and explicit code, solving immediately all conversion problems and facilitating the return from explicit dynamic to implicit continuation analysis. One of the problems with the current conversion from one element type to the other is the loss of information. The shell element used for the continuation analysis and the implicit time integrator is a modern shell element with full 6 degrees of freedom per node, including drill rotation. Also the through-thickness integration is performed analytically, resulting in improved performance and improved accuracy of the element [27]. The implementation of the Rebel shell element in the explicit code would be an interesting task and should greatly improve the mode-jumping results.

In order to obtain some useful results, a more computationally intensive approach was used where the transition from implicit static to

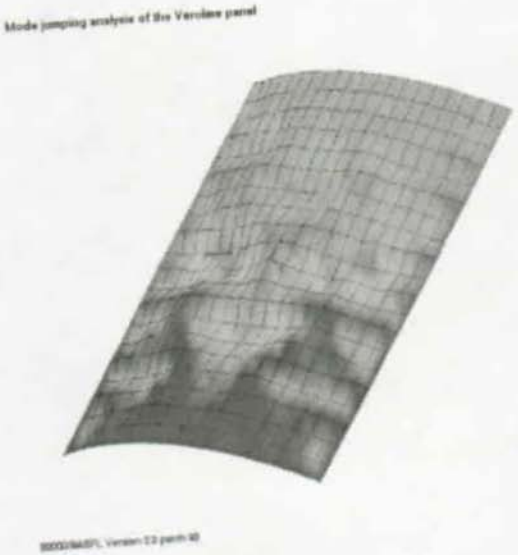


Figure 9.31: Fully dynamic buckling simulation with B2ETA

explicit dynamic was done in a much earlier stage of the continuation analysis. After only two steps with the continuation module the simulation was continued with the explicit method, where the panel was slowly compressed. This displacement was kept constant just beyond the limit point. The start from a static equilibrium was assured by assuming the stresses were almost zero due to the small initial deformation. This computation is quite time-consuming due to the long integration period, but it yields some very interesting results.

The first simulation showed a converged solution which is approaches the stable solution found with the implicit code, as shown in figure 9.31. This shows that the explicit code is capable of performing the mode jumping simulation and obtains a results which resembles the actual solution. The simulation also showed that the explicit code briefly follows

the quasi-static pre-buckling path. After a while, however, the dynamic effects take over. This could also explain why the explicit code reaches a static state before static equilibrium has been reached: the Rayleigh damping in the code absorbs the energy otherwise stored in the deformation of the structure.

9.3.5 Discussions on the behaviour of the explicit code

The results of the explicit mode-jumping simulations have been thoroughly discussed with two of the co-rapporteurs of this report, Dr. C.C. Rankin and Dr. E. Riks, both being experts in this field. The final conclusion of these discussions was, that the problem of the bad results have to be found in the performance of the Belytschko-type shell element for the following reasons:

1. The incremental method for integrating the stresses introduces too large errors in the solution for this type of simulations. A direct Lagrangian stress definition would avoid this problem.

2. The use of reduced integration for the shell element introduces additional errors, which deteriorate the solution.

3. The shell theory used for the element is not advanced enough to get good results for this type of simulations.

This all points to the same direction: In order to obtain better results for mode-jumping simulations (and probably also for other problems) a new, more advanced, Lagrangian shell element needs to be developed and implemented.

9.3.6 Conclusions

It has been shown here that it is indeed possible to perform mode-jumping simulations with an explicit finite element code instead of an implicit

one. This is useful thanks to the potential gains in computation time. The simulation of the Verolme panel ran 3 to 5 times faster with the explicit code then with the implicit code on a single processor. The explicit code also has the advantage of being very scalable on parallel computers, as shown in chapter 8, improving this advantage even further.

However, in order to be able to run the simulation properly and to prevent the loss of information due to convergence, which makes the conversion back to the implicit element difficult, it is highly recommended to implement the implicit element in the explicit code. With an element based on the element in the implicit code, future research in mode-jumping behaviour could profit from the use of explicit transient analysis.

9.4 Conclusions

In this chapter three applications of the explicit finite element code have been described. They all contain new aspects, from new material models to new applications. Because time was limited, they only show the possibilities of the application, but each of them requires continued investigation and development.

Optimisation

The use of the explicit finite element code for optimisation of material parameters showed its usefulness for finding the material parameters of the Linköping UD damage model. The possible applications of explicit analysis combined with an optimisation procedure are numerous, ranging from the design of small bars under impact load to the design of complete structures.

Impact

The simulation of a gelatine cylinder on a composite plate showed the capabilities of both the material damage model and the discrete element

method for gelatine. However, more work needs to be done in this field. First of all the proper parameters of the inter-particle law need to be obtained by simulating impact tests on a wall and comparing the results with actual tests. Secondly, with these proper parameters the simulations need to be performed again. Also other material models should be compared in order to judge the additional importance of the Linköping damage model. In addition, the mesh needs to be adapted with a finer mesh near the clamps in order to improve the results with fracture near the fixed ends. Finally, this is the kind of application which could potentially benefit the most from parallel computation. Therefore, the gelatine impact simulation example could be used to continue development of parallelism, including load balancing and automatic mesh decomposition.

Mode jumping

The use of the explicit finite element method for mode-jumping is promising, but requires some development to obtain correct results. The first thing to develop is a shell element based on the same shell theory which is used in the implicit code for the non-linear analysis up to the limit point. This would not only facilitate the conversion from one simulation type to the other, but would greatly improve the dynamic simulation itself. Also other non-linear elements could be implemented to perform this type of analysis. Only when other non-linear elements are introduced, conclusions can be drawn as to the actual usefulness of the explicit method for mode-jumping simulations.

10

Conclusions and recommendations

The main goal of this thesis has been the development of an object-oriented structure for a finite element code in order to improve maintainability and facilitate new developments. Contrary to similar developments found in literature, this new code structure should allow existing computational routines to be included, and the resulting executable should not be substantially slower than the existing sequential version of the code. The second goal is to modify the program in such a way that it can be run on high-performance parallel computers. The increasing demands for dealing with ever more complex problems outpace the development of computer hardware, thus making the use of parallel computing indispensable solving such problems in a reasonable amount of time. The object-oriented code should allow for this parallelism to be implemented transparently. An explicit finite element program was chosen for implementation the developments of object-oriented structure and parallelism. The nature of the explicit code, with its data and instructions closely coupled for efficiency reasons, makes it a suitable program which requires

the improved maintenance. The kind of problem usually solved with this type of program require large amounts of computation time, making parallel computation a requisite for modern explicit finite element codes.

Object-oriented program

The idea of object-oriented programming is the encapsulation of data in so-called objects. This way the local data is protected from accidental modification outside the object. As the risk of errors with global consequences is thus reduced, program maintainability is consequently greatly improved. Another feature of object-oriented programming is inheritance, i.e. allowing one object to obtain the characteristics, data and/or interface of another object. This reduces coding time and allows for the creation of one single interface for several different objects. By creating a program structure with the natural division in structural finite element methods of subdomains with nodes and elements, the logical setup of an object-oriented finite element code is created. The idea proposed in this thesis is to group all elements of the same type together in one object instead of treating each element individually. The advantage of this approach has been demonstrated in this thesis:

- The grouping of elements significantly reduces overhead, thus resulting in faster executables, while maintaining the object-oriented structure of the code. The same is the case for the grouping together of all nodes.

- This idea also corresponds more closely to the existing data structure of the old finite element program. The result is an object-oriented core wrapped around existing Fortran and C computational routines. The main advantages are the ability to create an object-oriented code without having to rewrite all the existing routines and a resulting executable just as efficient as the original Fortran program.

- Collecting elements by element type allows for more flexibility in

the introduction of new element types as demonstrated by the implementation of the discrete element method.

Parallel program

The object-oriented finite element code allows for easy and transparent implementation of parallelism. The creation of more or less independent subdomain objects makes the introduction of parallelism using domain decomposition easy. The communication between the subdomains can be implemented in a communication class which is inherited by the subdomain class. This way the communication is introduced between the subdomains without the developer being disturbed by it or even noticing it. No knowledge of parallel computing is required to continue development of the explicit finite element code. For the parallel computation itself it has been shown that:

- The use of domain decomposition allows for a straightforward and natural implementation of parallelism in an explicit finite element code.
- Given a properly balanced computation, the explicit finite element code is very well suited for parallel execution.
- The parallel performance depends on the way the domain has been decomposed and on the specifications of the hardware. When the processor performance and network performance are known, an analysis of a specific problem and the decomposition of the mesh of this problem will give a good indication of the parallel performance on this given architecture and for the particular problem.

Applications

The applications in this report show the possibilities of the explicit finite element code. They are meant as an indication for future research and development. They demonstrate that:

- The explicit finite element can be used for structural optimisation. This allows for the optimisation of material parameters for complex material models. Other optimisation calculations should also be possible.

- The simulation of gelatine impact is a complicated and time consuming task. Of all the applications shown, this is the one which will mostly benefit from parallel computing. The use of parallel computers for this type of simulation not only reduces the total simulation time, but also allows for the creation of more detailed models needed to obtain correct solutions.

- The explicit finite element program can be used in combination with an implicit finite element program for the simulation of buckling behaviour and the mode jumping phenomenon. For this program combination to work properly, the explicit code should have a similar element to the one in the implicit code, so that no conversion is necessary.

Recommendations

The developments described in this thesis provide a good starting point for continued research in different areas. The object-oriented explicit finite element program has shown to perform well and could be used with confidence for future developments. As far as the programming and the code structure itself are concerned, the following points need further investigation:

Program development

- Application of the object-oriented wrapper to an implicit finite element code. The principle of wrapping existing sequential code in an object-oriented structure has been shown to work for an explicit finite element code. Implementing the principle of wrapping

procedural code in an implicit finite element code will be a more laborious task due to the different computational parts in the program, but the larger complexity of the implicit code should benefit greatly from the improved maintainability.

- Improvement of the material data structure and its classes in order to create a more general and applicable set of classes. As discussed in chapter 6, different structures of the material classes can be envisioned. An investigation into the pros and cons of the different possibilities should be made in order to determine which type of implementation to use.

- Automatic or pre-processing mesh decomposition. With the current implementation, the user needs to decompose the mesh in multiple subdomains. In B2000 this is not a real problem, as more complicated meshes are generated by multiple patch commands. The input processor then creates not only the subdomains, but also computes the mesh connectivity required for communication. However, most commercial pre-processing tools, like Patran, and CAD tools, do not have the option for generating multiple subdomains. The decomposition of the mesh needs to be carried out in a pre-processing step before launching the computation.

- Static or dynamic load balancing for parallel computations. Due to changes in the load of a parallel computer, different processor speeds, sudden changes in the computational effort in an element (like plasticity) or simply the use of wrong weight factors for the load balancing can affect the proper balance of the parallel computation. This results in some processes waiting idly for other processes to finish. Research within this field could greatly improve the speedup of a parallel computation, especially in the case of complex industrial applications. The object-oriented setup of the code with independent subdomains should prove to be of great help.

Applications

The applications in chapter 9 are a demonstration of the possibilities of the explicit finite element program. They also point to continued research on the use of explicit finite element methods:

- Application of the new explicit finite element program to large industrial problems. The applications and tests shown in this thesis are relatively small models and often academic in nature. For the usefulness of the program and the effectiveness of its implementation, large scale industrial applications should be analysed.

- Obtain properly inter-particle law parameters for gelatine, investigate sensibility of parameters, and influence of particle size. The gelatine model as developed in this thesis needs to be properly evaluated. Unfortunately, due to time constraints such and evaluation had to be left out here.

- Continue the investigation of the possibilities of structural optimisation using the explicit finite element method. The optimisation of material parameters is only one possible application of the optimisation tool. Its true value will be in the optimisation of the dynamic behaviour of structures, like in the design if impact bars or in the optimisation of crashworthiness.

- The study of impact on structures is a research field on its own and worthy of investigation. This study could be applied to all fields of structural engineering, including civil engineering, aerospace engineering, automobile and train design or bio-mechanics.

- In order to obtain good results for mode jumping simulations the following tasks need to be done:

  - Implementation of a shell element with 4-point Gauss integration, preferably based on the same element used in the implicit code. This will give the explicit finite element code

a shell element with improved accuracy, at the same time avoiding the problems of conversion between the implicit and the explicit program.

- Mode jumping simulations using this new element in order to evaluate the response. For this, plenty of examples are available.
- Decomposition of the mesh in order to take advantage of parallel computers to reduce simulation time. This could be of particular importance for larger models.
- Study the behaviour of more complex models which include contact (e.g. composite panels with delamination).

The work presented in this thesis provides only a framework necessary for the continued development of finite element codes and its applications. The availability of a well designed program which allows the use of high-performance computers will open the way to new areas of research in structural analysis, to new applications, and to more detailed and more accurate analysis of mechanical structures.

Bibliography

[1] ALMASI, G. S., AND GOTTLIEB, A. *Highly Parallel Computing.* Benjaming/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[2] BATHE, K. *Finite element procedures in engineering analysis.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1982.

[3] BELYTSCHKO, T. *An overview of semidiscretization and time integration procedures.* In Belytschko and Hughes [4], 1981, pp. 1–65.

[4] BELYTSCHKO, T., AND HUGHES, T., Eds. *Computational methods for transient analysis.* Elsevier Science Publishers B.V., Amsterdam, 1981.

[5] BESSON, J., AND FOERCH, R. Large scale object-oriented finite element code design. *Computer methods in applied mechanics and engineering 142* (1997), 165–187.

[6] DUBOIS-PÈLERIN, Y., AND ZIMMERMANN, T. Object-oriented finite element programming: II. A prototype program in Smalltalk. *Computer methods in applied mechanics and engineering 98* (1992), 361–397.

[7] EDLUND, U. A UD composite ply failure model. In *Impact and damage tolerance modelling of composite materials and structures*

(1999), C. Soutis and I. Guz, Eds. Proceedings of the EUROMECH 400 colloquium, Imperial College, London.

[8] FLANAGAN, D., AND BELYTSCHKO, T. A uniform strain hexahedon and quadrilateral with orthogonal hourglass control. *International journal for numerical methods in engineering 17* (1981), 679–706.

[9] FRANSEN, R. W. Explicit transient structural response analysis: Report I: an exploration of techniques. Master's thesis, Delft University of Technology, Delft, The Netherlands, 1994.

[10] FRANSEN, R. W. Explicit transient structural response analysis: Report II: an implementation of techniques. Master's thesis, Delft University of Technology, Delft, The Netherlands, 1994.

[11] GROPP, W., LUSK, E., AND SKJELLUM, A. *Using MPI.* MIT Press, Cambridge, Massachusetts, 1996.

[12] HALLQUIST, J. LS-DYNA3D theoretical manual. Tech. rep., Livermore Software Technology Company, Livermore USA, 1993.

[13] HARDING, J., AND PETRINIC, N. Documentation on Oxford high-speed structure tests. Tech. rep., University of Oxford, Department of Engineering Science, 1999.

[14] HUGHES, T. *Analysis of transient algorithms with particular reference to stability behaviour.* In Belytschko and Hughes [4], 1981, pp. 67–155.

[15] JOHNSON, A. High velocity impact of composite aircraft structures HICAS. Tech. rep., DLR Stuttgart, 1997. BriteEuram 97-4238, Contract No. BRPR-CT97-0543.

[16] JOHNSON, A. In-plane damage models for fabric reinforced composites. Tech. rep., DLR Institute of Structures and Design, D-70569, Stuttgart, 1999. HICAS Report D.2.1.2.

[17] JOHNSON, A., AND PICKETT, A. Impact and crash modelling of composite structures. In *European conference on computational mechanics* (1999). ECCM99 Munich.

[18] JOHNSON, A., AND PICKETT, A. Numerical modelling of composite structures under impact loads. In *DYMAT 2000, Krakow* (2000). to be published.

[19] KIKUCHI, N., AND ODEN, J. *Contact problems in elasticity: a study in variational inequalites and finite element methods.* SIAM Publication, Philadelphia, 1988.

[20] LADEVEZE, P., AND LE DANTEC, E. Damage modeling of the elementary ply for laminated composites. *Composites Science and Technics 43* (1992), 257–267.

[21] MERAZZI, S. *Modular finite element analysis tools applied to problems in engineering.* PhD thesis, DGC-EPFL, Lausanne, Switzerland, 1994. Thèse No 1251.

[22] MERAZZI, S., AND STEHLIN, P. B2000 data structure and programming handbook. Tech. rep., SMR Engineering & Development, Bienne, Switzerland, 1994.

[23] MERAZZI, S., STEHLIN, P., AND DE BOER, A. B2000 processors reference manual. Tech. rep., SMR Engineering & Development, Bienne, Switzerland, 1995.

[24] MESSAGE PASSING INTERFACE FORUM. MPI: A message-passing interface standard. Tech. rep., University of Tennessee, Knoxville, Tennessee, 1995.

[25] PARK, K. An improved stiffly stable method for direct integration of nonlinear structural dynamic equations. *ASME Journal of Applied Mechanics 42* (1975), 464–470.

[26] PETRINIC, N. DEST - discrete element simulation tools; theory manual. Tech. rep., University of Oxford; Solid Mechanics Group, 1999.

[27] REBEL, G. *Finite rotation shell theory including drill rotations and its finite element implementation.* PhD thesis, Delft University of Technology, Delft, The Netherlands, 1998.

[28] REMMERS, J. J. C. Mode-jumping with B2000. Master's thesis, Delft University of Technology, Delft, The Netherlands, 1998.

[29] RIKS, E. The application of newton's method to the problem of elastic stability. *ASME Journal of Applied Mechanics 39* (1972), 1060–1066.

[30] RIKS, E., AND RANKIN, C. Computer simulation of the buckling behaviour of thin shells under quasi-static loads. *Archive of Computational Methods in Engineering 4* (1997), 325–351.

[31] RIKS, E., RANKIN, C., AND BROGAN, F. On the solution of mode jumping phenomena in thin-walled shell structures. *Computer Methods in Applied Mechanics and Engineering 136* (1996), 59–92.

[32] SCHILDT, H. *C++, The complete reference, second edition.* Osborne McGraw-Hill, Berkely, California, 1995.

[33] SIMO, J., AND HUGHES, T. *Computational inelasticity.* Springer Verlag, 1997.

[34] SNIR, M., ET AL. *MPI, the complete reference.* MIT Press, Cambridge, Massachusetts, 1996.

[35] STEIN, M. Loads and deformation of buckled rectangular plates. Tech. rep., National Aeronautics and Space Administration, 1959.

[36] STROUSTRUP, B. *The C++ programming language, 3rd edition.* Addison-Wesley, 1997.

[37] VEROLME, K. *The development of a design tool for fiber metal laminate compression panels.* PhD thesis, Delft University of Technology, Delft, The Netherlands, 1995.

[38] VOLGERS, P. T. G. Contact in finite elements. Master's thesis, Delft University of Technology, Delft, The Netherlands, 1997.

[39] VOLGERS, P. T. G. Introduction to programming in B2000. Tech. rep., Delft University of Technology, Delft, The Netherlands, 1997.

[40] VOLGERS, P. T. G. Lagrange-like contact in ETA. Master's thesis, Delft University of Technology, Delft, The Netherlands, 1997.

[41] VOLGERS, P. T. G. ETA manual, C++ version. Tech. rep., IMAC-EPFL, Lausanne, Switzerland, 2000.

[42] ZHONG, Z.-H. *Finite element procedures for contact-impact problems.* Oxford University Press, Walton Street, Oxford, 1993.

[43] ZIENKIEWICZ, O., AND TAYLOR, R. *The finite element method, forth edition, Volume 1.* McGraw-Hill Book Company, Berkshire, U.K., 1994.

[44] ZIENKIEWICZ, O., AND TAYLOR, R. *The finite element method, forth edition, Volume 2.* McGraw-Hill Book Company, Berkshire, U.K., 1994.

[45] ZIMMERMANN, T., DUBOIS-PÈLERIN, Y., AND BOMME, P. Object-oriented finite element programming: I. Govering principals. *Computer methods in applied mechanics and engineering 98* (1992), 291–303.

Appendices

A

Theory of the 8 node volume element

The 8 node volume element `H8.ETA` is a uniform strain hexahedron as developed by Flanagan and Belytschko and described in [8]. Here follows a brief summary limited to the volume element only.

A.1 Kinematics

The isoparametric shape functions map a unit cube in ξ_i space (or explicity written as (ξ, η, ζ)) to a general hexahedron in x_i (or (x, y, z)) space. Choosing the centre of the unit cube as the origin in ξ_i space the shape functions can be written in terms of an orthogonal set of base vectors (see table A.1) as follows:

$$\begin{aligned}\phi_I = &\tfrac{1}{8}\Sigma_I + \tfrac{1}{4}\xi\Lambda_{1I} + \tfrac{1}{4}\eta\Lambda_{2I} + \tfrac{1}{4}\zeta\Lambda_{3I} + \\ &\tfrac{1}{2}\eta\zeta\Gamma_{1I} + \tfrac{1}{2}\zeta\xi\Gamma_{2I} + \tfrac{1}{2}\xi\eta\Gamma_{3I} + \tfrac{1}{2}\xi\eta\zeta\Gamma_{4I}\end{aligned} \tag{A.1}$$

The above vectors represent the displacement modes of the unit cube.

A.1 Kinematics

I	ξ	η	ζ	Σ_I	Λ_{1I}	Λ_{2I}	Λ_{3I}	Γ_{1I}	Γ_{2I}	Γ_{3I}	Γ_{4I}
1	$-\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{1}{2}$	1	-1	-1	-1	1	1	1	-1
2	$\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{1}{2}$	1	1	-1	-1	1	-1	-1	1
3	$\frac{1}{2}$	$\frac{1}{2}$	$-\frac{1}{2}$	1	1	1	-1	-1	-1	1	-1
4	$-\frac{1}{2}$	$\frac{1}{2}$	$-\frac{1}{2}$	1	-1	1	-1	-1	1	-1	1
5	$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	1	-1	-1	1	-1	-1	1	1
6	$\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	1	1	-1	1	-1	1	-1	-1
7	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	1	1	1	1	1	1	1
8	$-\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	-1	1	1	1	-1	-1	-1

Table A.1: Base vectors (I=node number)

The vector Σ_I is the rigid body translation, the vectors Λ_{iI} are the linear, uniform normal strain modes and the last four vectors $\Gamma_{\alpha I}$ can be identified as the linear strain modes which are neglected by the one-point integration. These vectors are called the *hourglass* base vectors as they describe the hourglass patterns for a unit cube. Note that we use the following notation: the capital index I denotes the nodal points and range from 1 to 8, the index i ranges from 1 to 3 and the Greek subscript ranges from 1 to 4.

Using the principle of virtual work, we get for the relation for the element nodal forces f_{iI}:

$$\dot{u}_{iI}\, f_{iI} = \int_V T_{ij} D_{ij}\, \mathrm{d}V \tag{A.2}$$

where $\dot{u}$ the velocity, T_{ij} the Cauchy stress tensor, D_{ij} the deformation rate tensor and V the volume. Using one-point integration, thereby neglecting the nonlinear part of the element displacement field, the expression is approximated by:

$$\dot{u}_{iI}\, f_{iI} = V \bar{T}_{ij} \dot{\bar{u}}_{i,j} \tag{A.3}$$

where $\bar{T}_{ij}$ and $\dot{\bar{u}}$ denote the assumed uniform stress field (called the mean stress tensor) and the mean velocity respectively. The mean kinematic

quantities are defined by integrating over the element as follows:

$$\dot{\bar{u}}_{i,j} = \frac{1}{V}\int_V \dot{u}_{i,j}\,\mathrm{d}V \tag{A.4}$$

The B-matrix is now defined as:

$$B_{iI} = \int_V \phi_{I,i}\,\mathrm{d}V \tag{A.5}$$

So that the mean velocity gradient is given by:

$$\dot{\bar{u}}_{i,j} = \frac{1}{V}\dot{u}_{iI}B_{jI} \tag{A.6}$$

Now we can express the nodal forces by:

$$f_{iI} = \bar{T}_{ij}B_{jI} \tag{A.7}$$

Computing the nodal forces now requires evaluation of the B-matrix and the volume. As $x_{i,j} = \delta_{ij}$ and $x_i = x_{iI}\phi_I(\xi,\eta,\zeta)$ we can substitute this into Eq.(A.5) to get:

$$x_{iI}B_{jI} = \int_V (x_{iI}\phi_I)_{,j}\,\mathrm{d}V = V\delta_{ij} \tag{A.8}$$

or

$$B_{iI} = \frac{\partial V}{\partial x_{iI}} \tag{A.9}$$

Using some extensive calculations (see [8]) we can find a way to compute the B matrix and from that the volume. These expressions are given in Section A.4.

A.2 Constitutive relations

The only stress-strain relation implemented so far is an isotropic elastic material. Because the integration scheme uses only the linear strain

rate terms, the stress rate cannot depend on the nonlinear portion of the displacement field. Hence, the mean stress must be related to the mean strain rates. The physical stress rate for such a material is described by:

$$\dot{T}_{ij} = \bar{T}_{ij}^{\nabla} + \bar{W}_{ik}\bar{T}_{kj} + \bar{W}_{jk}\bar{T}_{ki} \tag{A.10}$$

where $\bar{W}_{ij}$ the (mean) vorticity tensor and $\bar{T}_{ij}^{\nabla}$ the Jaumann rate, which obeys Hooke's law:

$$\bar{T}_{ij}^{\nabla} = \lambda \bar{D}_{kk}\delta_{ij} + 2\mu \bar{D}_{ij} \tag{A.11}$$

where $\bar{D}_{ij}$ the deformation rate tensor and λ and μ the Lamé coefficients. With this constitutive law we can calculate the eigenmodes and the eigenvalues of the hexahedron. According to [3] and [8] the maximum frequency ω_{max} is bounded by:

$$8\,\frac{\lambda + 2\mu}{\rho}\,\frac{B_{iI}B_{iI}}{V^2} \geq \omega_{max}^2 \geq \frac{8}{3}\,\frac{\lambda + 2\mu}{\rho}\,\frac{B_{iI}B_{iI}}{V^2} \tag{A.12}$$

where ρ the density. The central difference time integration scheme is stable if [14] [4]:

$$\Delta t \leq \frac{2}{\omega_{max}}\left(\sqrt{(1-\epsilon^2)} - \epsilon\right) \tag{A.13}$$

where ϵ the fraction of the critical damping in the highest frequency. Omitting the influence of the damping we obtain for the critical timestep for stability of the explicit scheme for the general hexahedron:

$$\Delta t \leq V\sqrt{\frac{\rho}{2(\lambda + 2\mu)B_{iI}B_{iI}}} \tag{A.14}$$

A.3 Anti-hourglassing

The constitutive relations combined with the integration scheme described above take only the linear portion of the displacement field into account,

causing certain forms of deformation to be energy-free. These are the so-called hourglass modes. In order to prevent instability due to these deformation modes dominating the deformation, the hourglass modes are treated separately.

Considering the total nodal velocity field $\dot{u}_{iI}$ to be made out of the linear portion $\dot{u}_{iI}^{lin}$ and the hourglass field $\dot{u}_{iI}^{hg}$, we can define the hourglass field by:

$$\dot{u}_{iI}^{hg} = \dot{u}_{iI} - \dot{u}_{iI}^{lin} \tag{A.15}$$

It can be shown [8] that the hourglass field is orthogonal to Σ_I and B_{iI} and the base vectors Λ_{iI} except the hourglass base vectors. Therefore we can expand $\dot{u}_{iI}^{hg}$ as follows:

$$\dot{u}_{iI}^{hg} = \frac{1}{\sqrt{8}} \dot{q}_{i\alpha} \Gamma_{\alpha I} \tag{A.16}$$

where the hourglass modal velocities are represented by $\dot{q}_{i\alpha}$ and the constant is added to normalise $\Gamma_{\alpha I}$. We now define the hourglass shape vector $\gamma_{\alpha I}$ such that:

$$\dot{q}_{i\alpha} = \frac{1}{\sqrt{8}} \dot{u}_{iI} \gamma_{\alpha I} \tag{A.17}$$

Using the equations above and following [8] we can compute $\gamma_{\alpha I}$:

$$\gamma_{\alpha I} = \Gamma_{\alpha I} - \frac{1}{V} B_{iI} x_{iJ} \Gamma_{\alpha J} \tag{A.18}$$

Note the difference between the hourglass base vectors $\Gamma_{\alpha I}$ and the hourglass shape vectors $\gamma_{\alpha I}$. For a general hexahedron, $\Gamma_{\alpha I}$ is orthogonal to B_{iI} and defines the hourglass pattern while $\gamma_{\alpha I}$ is orthogonal to the linear velocity field $\dot{u}_{iI}^{lin}$ and is necessary to detect the hourglassing.

There are different possibilities to treat the hourglass difficulties. Most common methods are the addition of artificial damping or the addition of artificial stiffness. The first approach has the disadvantage that hourglass deformation is permanent, since there is no stiffness in the modes. The second approach is more successful and is incorporated in the element.

To control the hourglass modes we define the generalised hourglass $Q_{i\alpha}$ conjugate to $\dot{q}_{i\alpha}$ so that the rate of work is given by:

$$\dot{u}_{iI} f_{iI}^{hg} = Q_{i\alpha} \dot{q}_{i\alpha} \tag{A.19}$$

Using Eq.(A.17) the hourglass nodal force is given by:

$$f_{iI}^{hg} = \frac{1}{\sqrt{8}} Q_{i\alpha} \gamma_{\alpha i} \tag{A.20}$$

Following [8] we get for the hourglass resistance in case of artificial stiffness:

$$\dot{Q}_{i\alpha} = \kappa \frac{\lambda + 2\mu}{3} \frac{B_{iI} B_{iI}}{V} \dot{q}_{i\alpha} \tag{A.21}$$

where κ the hourglass stiffness parameter. Tests show that a good value for κ is somewhere between 0.01 and 0.05.

A.4 Computation of the B-matrix and the volume

According to [8], App I, the first term of B_{iI} is:

$$\begin{aligned} B_{11} = & \frac{1}{12}[y_2((z_6 - z_3) - (z_4 - z_5)) + y_3(z_2 - z_4) \\ & + y_4((z_3 - z_8) - (z_5 - z_2)) + y_5((z_8 - z_6) - (z_2 - z_4)) \\ & + y_6(z_5 - z_2) + y_8(z_4 - z_5)] \end{aligned} \tag{A.22}$$

The other terms of the B-matrix can be obtained by permuting the nodes according to table A.2 and the coordinate axes by cyclic permutation. The volume can now be computed by contracting the B-matrix and nodal coordinates using Eq.(A.8) to yield:

$$V = \frac{1}{3}(x_I B_{1I} + y_I B_{2I} + z_I B_{3I}) \tag{A.23}$$

1	2	3	4	5	6	7	8
2	3	4	1	6	7	8	6
3	4	1	2	7	8	5	6
4	1	2	3	8	5	6	7
5	8	7	6	1	4	3	2
6	5	8	7	2	1	4	3
7	6	5	8	3	2	1	4
8	7	6	5	4	3	2	2

Table A.2: Nodal permutations

B

Damage models for composite materials

B.1 Material definition of UD damage model

The material properties in B2000 are defined by the EMAT keyword. The elements or laminates refer to the corresponding material properties by the material ID number *mid*. For better readability command keywords are printed in uppercase, although keywords are not case sensitive. Values are printed in italics.

Materials are defined consecutively from 1 to the number of materials. Gaps in the numbering must be avoided.

Synopsis

```
emat
  mid 1
    parameters
  endmid
```

```
  mid 2
    parameters
  endmid
endemat
```

Parameters

- MID *mid*
  Material identification number. All subsequent parameters refer to the material number *mid* until endmid is specified. Materials are defined consecutively from 1 to the number of materials. Gaps in the numbering should be avoided.

- TYPE *otho2*
  Refers to the material type. For UD plate material, this is a 2D orthotropic material.

- PLTY *link1*
  Refers to the plasticity type, in this case the UD damage model from Linköping number 1.

- THETA *val1 val2*
  Defines the plastic softening parameters θ_{22} and θ_{12}. (Note: $\theta_{11} \equiv 0$.)

- K0 *val*
  Defines the plastic yield limit κ_0.

- PLCF *val*
  Defines the plastic coupling coefficient a.

- PLVISC *val1 val2*
  Defines the plastic viscosity η_p and the exponent m_p.

- DAMAGESP *val1 val2 val 3*
  Defines the damage softening parameters βij.

- DAMAGEYL *val1 val2 val 3*
  Defines the damage yield limits l_{ij}.
- DAMAGECF *val*
  Defines the damage coupling coefficient b.
- DAMAGEVISC *val*
  Defines the damage viscosity η_d.
- DAMAGEXI *val1 val2*
  Defines the damage parameters ξ_1 and ξ_2.

B.2 Implementation tests of UD damage model

In order to check the correct implementation of the material model, some elementary tests were run directly on the subroutine and the finite element implementation. The results were then compared to analytical solutions obtained by U. Edlund.

B.2.1 Damage, non-viscous response in shear

The first test is the example as described in [7]. A pure shear loading situation is assumed, such that $\epsilon_{12} = c\,t$ where c is constant. Assuming no plastic flow takes place and that damage growth is initiated at $\epsilon_{12} = \epsilon_{12}^*$ and $\dot{\mu}_{12} > 0$, one finds for the damage parameter d_{12}:

$$d_{12} = 2\frac{b}{\beta_{12}} G_{12} \left({\epsilon_{12}}^2 - {\epsilon_{12}^*}^2 \right) \tag{B.1}$$

Substituting the following values: $\epsilon_{12}^* = XX$, or $l_{12} = 0.54$, $G = 1000$, $b = 0.3$ and $\beta_{12} = 5.0$ we get at $\epsilon_{12} = 0.05$ the results as summarised in Table B.2.1.

As can be seen from Table B.2.1 the Euler scheme converges linearly towards the analytical solution.

B.2.2 Damage, viscous response in shear

The second test is similar to the first one, but now the response is viscous, or rate dependent. The analytical solution is obtained with the help of the computer program Maple. By assuming $\langle g_2 \rangle = 0$ and $\langle g_{12} \rangle = g_{12}$ the following analytical expression for the damage parameter d_{12} can be found:

$$d(t) = \frac{2}{b\beta_{12}^3} G_{12} c^2 \left[b^2 \beta_{12}^2 t^2 - 2\eta_d b \beta_1 2t + 2\eta_d^2 - 2\eta_d^2 e^{-b\frac{\beta_{12}}{\eta_d} t} \right] \tag{B.2}$$

For the numerical test the following values are used: $G = 1000$, $\nu_{12} = 0.3$, $b = 0.3$, $\eta_d = 0.01$ and $l_{12} = 0.0$. This yields at $t = 0.01$, $\epsilon_{12} = 0.1$, for the damage parameter $d_{12} = 0.42866$ and the stress $\sigma_{12} = 114,268$. The numerical results are shown in Table B.2.2. The FE simulation results are also plotted for different values of the strain rate. Figure B.1 shows the stress-strain curve and Figure B.2 shows the development of the damage parameter d_{12}.

B.2.3 Plastic, non-viscous response in shear

The third test assumes that no damage takes place, but only plastic deformation of the matrix. Once again, only shear loading is considered. With $\theta_{22} = \theta_{12} = \theta$ and $\kappa_0 = 0$ a constant increasing stress-strain curve is

method	Δt	d_{12}	σ_{12}
analytical	-	0.192	80.8
routine ($c = 500$)	1.E-7	0.216536159	78.4247314
routine	1.E-8	0.192616750	80.7463983
routine	1.E-9	0.192001991	80.8006066
element B2000	8.E-10	0.191798	80.8199

Table B.1: Results of UD material, test 1

method	Δt	d_{12}	σ_{12}
analytical	-	0.42866	114.268
routine ($c = 10$)	1.E-6	0.428669244	114.267295
routine	1.E-7	0.428661994	114.267715
element B2000	8E-10	0.428661059	114.267781

Table B.2: Results of UD material, test 2

obtained,

$$\frac{\sigma_{12}}{\epsilon_{12}} = \frac{2G_{12}\,\theta}{2G_{12} + \theta} \tag{B.3}$$

With the values $G_{12} = 1000$ and $\theta = 100$, we get the numerical results as shown in Table B.2.3. (Results at $\epsilon = 0.01$) As shown in the table, the

method	Δt	η_*	$\sigma_{12}/\epsilon_{12}$
analytical	-	∞	95.238
routine ($c = 10$)	1.E-9	1.E-5	95.2471467
routine	1.E-9	1.E-6	95.2396428
element B2000	8.E-10	1.E-5	95.32666
element B2000	8.E-10	1.E-6	95.24596

Table B.3: Results of UD material, test 3

test requires a very small time step in order to obtain convergence. This is due to the value of the viscosity parameters η_p and η_d. Decreasing these parameters increases accuracy, but requires a smaller time step in order to maintain stability.

B.2.4 Plastic, viscous response in shear

The forth test is the same as test 3, but now with a viscous response. The resulting differential equation can be written as:

$$\dot{\epsilon}_{12}^{ir} + \frac{2G_{12}}{\eta_p}\epsilon_{12}^{ir} = \frac{2G_{12}}{\eta_p}\epsilon_{12} - \frac{\kappa_0}{\eta_p} \tag{B.4}$$

If we take $\kappa_0 = 0$ (i.e. $\epsilon_{12}^* = 0$) an expression can be found for the irreversible strain ϵ_{12}^{ir}:

$$\epsilon_{12}^{ir} = \frac{c}{2G_{12}}\left[2G_{12}t - \eta_p + \eta_p\, e^{-2\frac{G_{12}}{\eta_p}t}\right] \tag{B.5}$$

Using the following values: $G = 1000$, $c = 10$ and $\eta_p = 10$ with $\kappa_0 = 80$ (i.e. $\epsilon_{12}^* = 0.004$), we get for the irreversible strain at $t = 0.01$ ($\epsilon_{12} = 0.1$) the results as shown in Table B.2.4. The analytical solution is obtained by substituting the values in the differential equation and solving the resulting equation with Maple.

method	Δt	d_{12}	σ_{12}
analytical	-	0.0250597	149.8806
routine ($c = 10$)	1.E-7	0.0250595307	149.880941
routine	1.E-8	0.0250596921	149.880615
element B2000	8.E-10	0.0250597	149.881

Table B.4: Results of UD material, test 4

B.2.5 Viscous damage in fibre direction

The last test considers tension in the fibre direction. It is assumed that no plastic deformation takes place and that there is no stiffness in the direction perpendicular to the fibres. Solving the resulting differential

equation yields for the damage parameter d_1 in fibre direction:

$$d_1(t) = \frac{E_1 c^2}{2{\beta_1}^3}\left[(\beta_1 t)^2 - 2\eta_d\beta_1 t + 2{\eta_d}^2 - 2{\eta_d}^2 e^{\left(-\frac{\beta_1}{\eta_d}t\right)}\right] \quad \text{(B.6)}$$

With the following parameters: $E_1 = 1000$, $\beta_1 = 5.0$, $c = 10$, $\eta_d = 0.05$, $l_1 = 0$, we get at $t = 0.01$ for the damage parameter d_1 and the stress in the fibre direction σ_{11} the results as shown in Table B.2.5. For the numerical algorithm, setting $E_2 = 0$ and $\nu_{12} = 0$ is impossible due to division by zero. Therefore, small values are chosen, such that $E_2 = 1$ and $\nu_{12} = 0.01$.

method	Δt	d_1	σ_{11}
analytical	-	0.26424	-
routine ($c = 10$)	1.E-7	0.26427894	73.5769808
element B2000	8.E-10	0.279485	71.9401

Table B.5: Results of UD material, test 5

B.2.6 Figures of UD-material tests

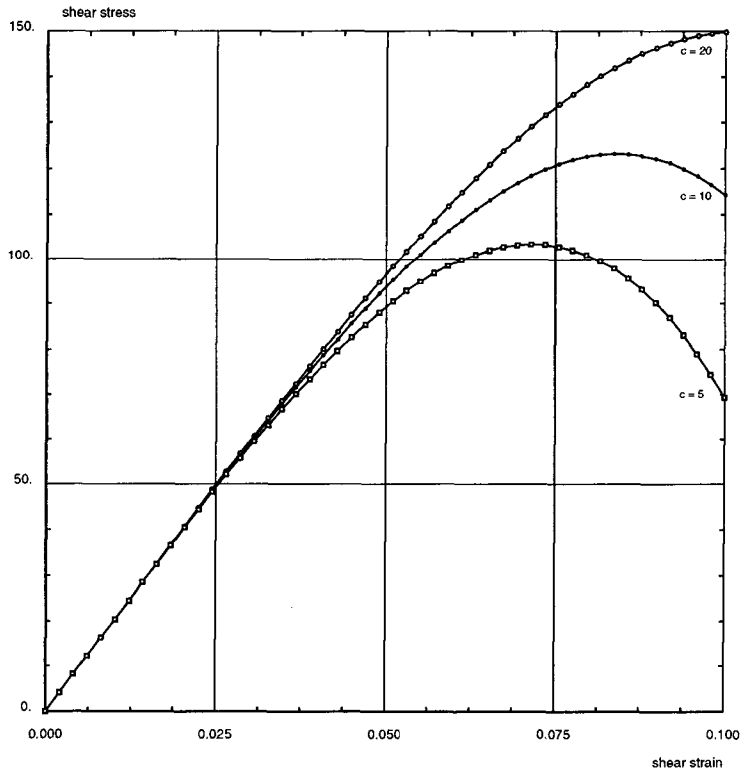


Figure B.1: Stress-strain curve for increasing strain rate. $\epsilon_{12} = ct$, for $c = 5$, $c = 10$ and $c = 20$.

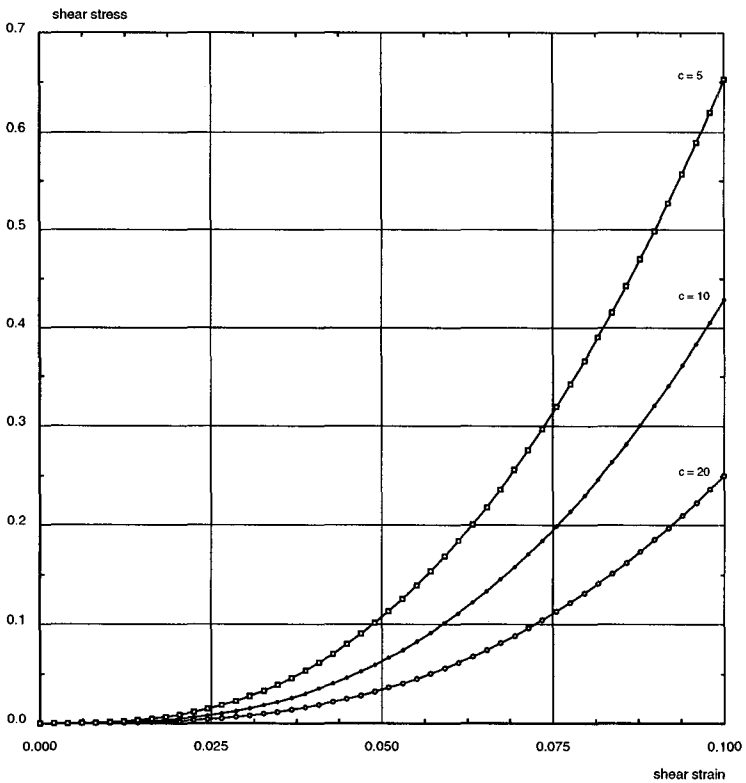


Figure B.2: Damage-strain curve for increasing strain rate. $\epsilon_{12} = ct$, for $c = 5$, $c = 10$ and $c = 20$.

B.3 Material definition of fabric damage model

Parameters

- MID *mid*
  Material identification number. All subsequent parameters refer to the material number *mid* until `endmid` is specified. Materials are defined consecutively from 1 to the number of materials. Gaps in the numbering should be avoided.

- TYPE ORTHO2
  Refers to the material type. For plates, this is a 2D orthotropic material.

- PLTY DLR1
  Refers to the plasticity type, in this case the fabric damage model from the DLR, number 1.

- ALPHA *val1 val2 val3*
  Defines the damage slope parameters α_1^t, α_1^c and α_{12}.

- DAMAGEYL *y1t y1c y12*
  Defines the damage yield values $Y_1^{0^{t,c}}$ in tension and compression and Y_{12}^0 for shear.

- DAMAGELIM *y1t y1c y12*
  Defines the maximum damage force values $Y_1^{f^{t,c}}$ in tension and compression and Y_{12}^0 for shear.

- PLYL *val*
  Defines the plastic yield limit R_0.

- PLVISC *val1 val2*
  Defines the plastic parameters β and the exponent m.

B.4 Implementation tests of fabric model

To test the implementation of the fabric model, tensile and shear tests have been simulated based on the material data for glass fiber/epoxy. The numerical results were compared to the figures found in [16].

B.4.1 Tensile test

The first test is a simulation of the standard tensile test in one of the fiber directions. The parameters used are found in [16] for glass fiber/epoxy and are shown in Table B.4.1. The obtained stress-strain curve, Figure

Material	Load	E_1 (GPa)	α_1 ($\sqrt{\text{MPa}^{-1}}$)	Y_1^0 ($\sqrt{\text{MPa}}$)	Y_1^f ($\sqrt{\text{MPa}}$)
Mat. 3 GF/epoxy	0^0-tension	30.8	0.072	1.0	4.0

Table B.6: Material parameters for fabric, elastic model

B.3, as well as the development of the damage parameter, Figure B.4, show very good results.

B.4.2 Shear test

To test the response of the model in shear, including the plastic behaviour, a cyclic simulation was carried out, like the one described in [16]. The additional material parameters are shown in Table B.4.2. The result of this simulation is shown in the stress-strain curve, Figure B.5. The trend corresponds to the test, but does not capture the hysteresis phenomena seen in the test. However, the proper plastic strain is computed, which continues to grow only after the previous maximum stress level has been reached. Figure B.6 shows the development of the damage parameter

versus the plastic strain. This also is in agreement with the results shown in the theory report.

Load	G_{12} (GPa)	α_{12} ($\sqrt{\text{MPa}^{-1}}$)	Y_{12}^0 ($\sqrt{\text{MPa}}$)
45^0-shear	6.38	0.213	0.144
Y_{12}^f ($\sqrt{\text{MPa}}$)	R_0 (MPa)	β (MPa)	m
5.9	12.8	17830	0.81

Table B.7: Material parameters for fabric, plastic shear model

B.5 Figures of fabric material tests

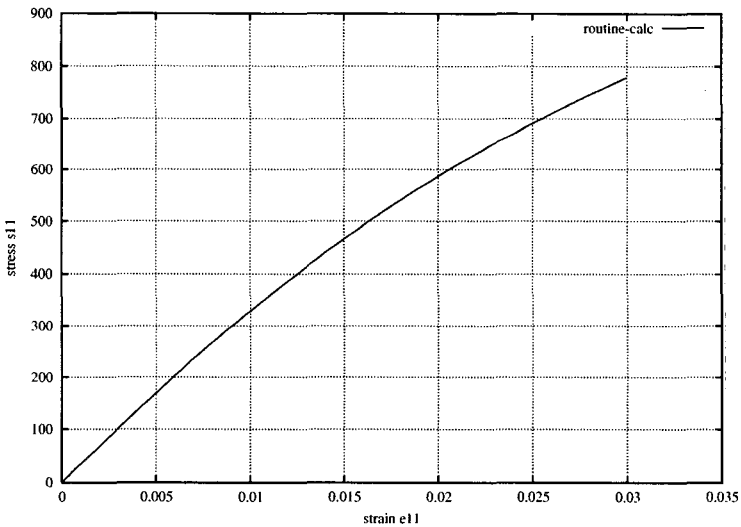


Figure B.3: Stress-strain curve in tensile test.

B.5 Figures of fabric material tests

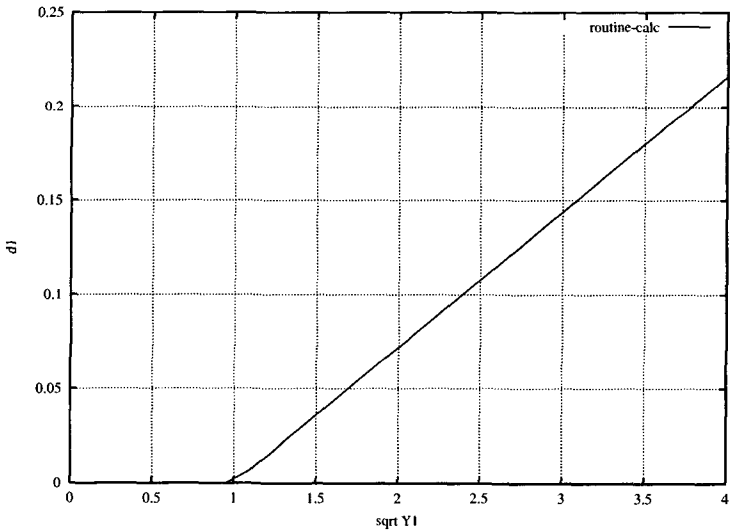


Figure B.4: Development of damage parameter in tensile test.

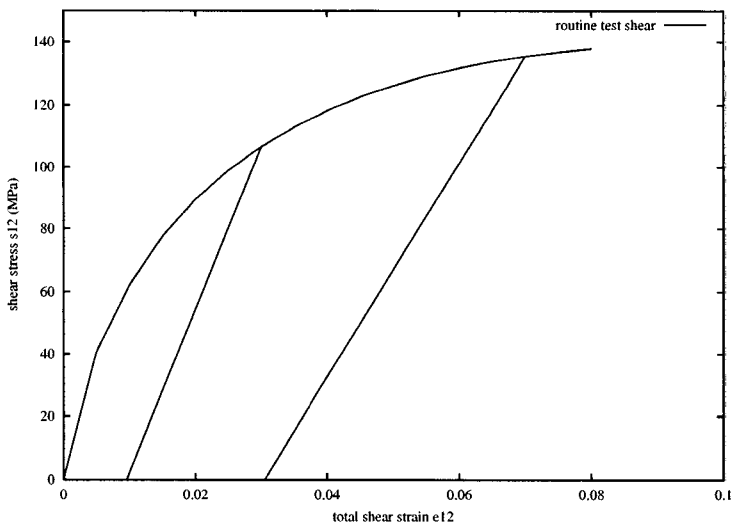


Figure B.5: Stress-strain curve in cyclic shear test.

B.5 Figures of fabric material tests

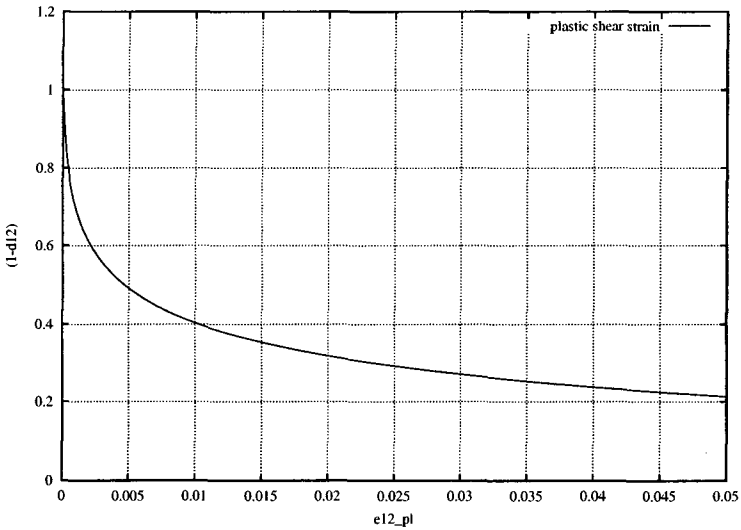


Figure B.6: Development of the damage parameter with the plastic strain.

C

Program manuals

C.1 Explicit Transient Analysis

C.1.1 Compilation

Two different versions of the B2ETA code can be compiled by adding or removing a compilation directive, B2MPI. Omitting this directive results in a executable only running in serial. Due to the nature of the implementation, this code is not capable of treating contact[1]. With the B2MPI-directive included, two environment variables need to be set, MPI_HOME and MPI_ARCH, denoting the place and architecture of the MPI implementation. Making use of MPI (Message Passing Interface) allows the executable to be run in parallel on shared or distributed memory computers.

[1] This is a matter which is still under development. Unfortunately, the current implementation does always require MPI-communication. Suggestions for a different implementation of the contact which can be run in parallel are welcome.

C.1.2 B2ETA - serial version

Due to the C/C++ main of the new code, the introduction of options and additional commands is no longer using the PCL-commands, but are done by Unix-like command line options. Typing `b2eta.x -help` will give you the syntax with a list of possible commands:

```
B2ETA: Explicit Transient Analysis Processor

ETA command line syntax:

Usage: b2eta database [options]
  where available options include:
  -help          :brings up this help screen
  -np   <NP>     :number of output plots   (np>0)
  -sf   <SF>     :timestep scalefactor     (0<sf<1)
  -ts   <TIME>   :simulation starttime     (ts>0)
                  starttime by default set to 0.
  -te   <TIME>   :simulation endtime       (te>ts)
  -plot <TYPE>   :output plot type         (1,2 or 3)
  -res  <NP>     :restart from output      (res>0)
                  restart by default set to 0.
  -end           :denotes the end of the options.
```

The last option `-end` might be required on certain platforms, but can in general be omitted. The database given is used as both the archival and the computational database. This might change in the future, especially if very large cases have to be computed. If no database is given, the default `test.db` will be tried and used if found.

C.1.3 B2ETA - MPI version

In order to run the explicit transient analysis processor in parallel, the mesh needs to be composed in the input file or by other means and put on the database in B2ETA readable format. For more on the B2000 input processor B2IP see [23, 41]. The MemCom data manager server, ARGUS, needs to be launched on the machine and in the directory where the

database is located. Then B2ETA needs to be started using the MPIRUN command:

```
% mpirun -np np $B2BIN/b2eta.x database [options]
```

where np the number of processes, `$B2BIN` the directory where the executable is located, `options` as described in C.1.2 and `database` as follows:

```
[username]@[machinename]:/{PATH}/database
```

where *username* the login name for the machine `machinename` where the database is located and `PATH` the full path name of the file.

C.2 Mode jumping simulations

The mode jumping simulations are performed in multiple stages: the continuation analysis and the dynamic simulation. Using the explicit dynamic analysis an additional conversion is required. The continuation module of B2000, B2CONT, is run using a script, which looks as follows:

```
#! /bin/csh -f
rm -rf panel.cdb
cp -rf panel.db panel.cdb

$B2000/bin/$B2ARCH/b2cont.x <<\
ar panel.db
co panel.cdb
go
\
```

where `panel.db` the database created by the input processor. Additional optional commands can also be introduced, see [23]. The resulting output from the continuation module is stored in a global displacement vector for each iteration. This information needs to be reconverted to the original numbering. This is done by the element branch vector processor B2EBV. The steering of this is similar as of the continuation processor. However, the information from this processor is not sufficient to restart with the explicit module. For that a conversion program has been written, B2I2E, to generate this information. Say we want to restart from the sixth iteration, the conversion program is run as follows:

```
$B2000/bin/$B2ARCH/b2i2e.x panel.cdb -cycle 6
```

Now the dynamic simulation can be run with B2ETA starting from the sixth iteration:

```
$B2000/bin/$B2ARCH/b2eta.x panel.cdb -res 6
```

Additional commands can be passed along as described in C.1.2.

C.3 Output to disk

The explicit transient analysis generates a substantial amount of data, which is stored on the database. B2ETA has three levels of output: BASIC, containing only the basic information on the dynamic behaviour of the structure; RESTART, containing additional information required to restart the computation; and COMPLETE, writing all information to disk, A list of all these datasets and their content is given here. All global datasets are named according to the syntax `setname.branch.output`, data related to elements are named `setname.0.type.branch.output`.

C.3.1 Basic output

- `GLVA.branch.output`: Global variables.
  Dataset containing the simulation time. The descriptor contains information like: time step, kinetic energy, strain energy, cycle, etc.

- `DISP.branch.output`: Displacements.
  Contains the nodal displacements for all degrees of freedom (3 or 6 per node).

C.3.2 Restart output

Contains all data described above and:

- `COOR.branch.output`: Coordinates.
  Contains the coordinates of all nodes.

- `VELO.branch.output`: Velocities.
  Contains the velocities of all nodes for all degrees of freedom.

- `DIRC.branch.output`: Direction normals.
  Contains the direction normals on the nodes. This information is also available in the dataset `DISP.branch.output`.

- Element output per element type:
  - `STRS.0.etype.branch.output`: Stresses.
    Contains all the computed stresses in all the integration points of the elements of type `etype`. For elements which store stresses only.
  - `HRGS.0.etype.branch.output`: Hourglass stresses/forces.
    Contains the values for the hourglass stresses or forces of the elements. For elements with under-integration only.
  - `STRN.0.etype.branch.output`: Stresses.
    Contains all the computed strains of the elements of type `etype`. For elements which store strains only. (Type 28)
  - `DAMG.0.etype.branch.output`: Damage variables.
    Contains the values of the damage variables array for the elements. Data contained in this array may vary from element type and from material model. (Currently 28 only.) For the material models described in this report the data for each *integration point* is: d(1-3): irreversible strain; d(4-6): damage values; d(7-9): plasticity values.
  - `DTAG.0.etype.branch.output`: Damage tag.
    Contains a tag for each element if element is broken or not. A tag greater than zero means the element completely is broken and does no longer participate in the analysis.

C.3.3 Complete output

Contains all data described above and:

- `FINT.branch.output`: Internal forces.
  Contains the internal forces on all nodes for all degrees of freedom.

D

Benchmark input file

```
title 'Beam, 64000 elements, 64 subdomains'

(N1=21)
(N2=11)
(N3=6)
(iclamped=N1*N2)
(ifree1=1+N1*N2*(N3-1))
(ifree2=N1*N2*N3)
(ibr=1)
(maxbr=64)
(offset=64/maxbr)
(xbegin=0)
(xend=xbegin+offset)

DYNA
  TIME_END 0.01
  NPLOTS 10
  SCALEFACT 0.8
  F 1  1E-9  0.0  2E-6  1.0  1.0  1.0  0.5  1.0
  SHEAR_CORRECTION 0.8
  HGMEMBRANE 0.04
```

```
  HGBEND 0.04
  HGSHEAR 0.004
  TIME_CALC
END
!
! branch on clamped end
!
branch=(ibr)
  patch nolist
    geom CUBE
    nn1 (N1) nn2 (N2) nn3 (N3)
    type HE8.ETA
    mid 1 nint 8
    p1 0. 0. 4.
    p2 0. 0. 0.
    p3 0. 2. 0.
    p4 0. 2. 4.
    p5 (xend) 0. 4.
    p6 (xend) 0. 0.
    p7 (xend) 2. 0.
    p8 (xend) 2. 4.
    end
!
  presvelo
     Velo=0.0
     Norm 1.0 0.0 0.0
     TRAN 1/(iclamped)
     Norm 0.0 1.0 0.0
     TRAN 1/(iclamped)
     Norm 0.0 0.0 1.0
     TRAN 1/(iclamped)
  end
!
endbranch
!
! middle branches
```

```
!
(ibr=ibr+1)
while (ibr<maxbr) (

(xbegin=xbegin+offset)
(xend=xend+offset)
branch=(ibr)
patch nolist
    geom CUBE
    nn1 (N1) nn2 (N2) nn3 (N3)
    type HE8.ETA
    mid 1 nint 8
    p1  (xbegin)  0.  4.
    p2  (xbegin)  0.  0.
    p3  (xbegin)  2.  0.
    p4  (xbegin)  2.  4.
    p5  (xend)  0.  4.
    p6  (xend)  0.  0.
    p7  (xend)  2.  0.
    p8  (xend)  2.  4.
    end
endbranch
(ibr=ibr+1)
!
)
!
! free end branch with initial velocity
!
branch (maxbr)
  patch nolist
    geom CUBE
    nn1 (N1) nn2 (N2) nn3 (N3)
    type HE8.ETA
    mid 1 nint 8
    p1 (xend) 0. 4.
    p2 (xend) 0. 0.
```

```
    p3 (xend) 2. 0.
    p4 (xend) 2. 4.
    p5    64. 0. 4.
    p6    64. 0. 0.
    p7    64. 2. 0.
    p8    64. 2. 4.
    end
!
  initvelo
    Velo = 1000.
    Norm 0.0 0.0 1.0
    TRAN 1156/(ifree2)
   end

endbranch
!
! isotropic material
!
ematerial
  mid 1
    type iso e 21000.0e6 p 0.0  dens 2500.
    endmid
  end
!
! automatically generate connectivity list
!
join
  auto
end
!
notopo
go
```

E

List of colleagues

- **ir. Paul Arendsen**
  Dutch Aerospace Laboratory (NLR)
  Department of Structures
  Vollenhove, the Netherlands
- **dr. Ulf Edlund**
  Linköping University
  Department of Mechanical Engineering
  Linköping, Sweden
- **dr. Alastair Johnson**
  German Aerospace Laboratory (DLR)
  Department of Structures and Materials
  Stuttgart, Germany
- **dr. Silvio Merazzi**
  SMR SA
  Bienne, Switzerland
- **dr. Nikica Petrinic**
  University of Oxford
  Department of Engineering Science
  Oxford, United Kingdom

- **ir. Joris Remmers**
  Delft University of Technology
  Department of Aerospace Engineering
  Delft, the Netherlands

Acknowledgements

The research presented in this thesis could not have been done without the help and contributions of many people. Above all I would like to thank my supervisor Dr. Silvio Merazzi for all his help, the discussions and everything else. Your optimism and sarcasm, your enormous expertise and visions which are sometimes ahead of their time, maybe we brought them a bit closer. I'll never regret coming here, working with you is a great pleasure.

I would also like to thank Prof. L. Pflug for giving me the chance to work here, for his faith in having me do the work I've done and for the inspiring discussions, also on subjects besides engineering.

I owe much gratitude to Dr. Eduard Riks for getting me into finite elements in the first place, for the discussions over the years before and during my thesis and for all the rest. If it wasn't for you I would not have been here in the first place.

Many thanks to all the people I have worked with and who contributed directly with their ideas, theory or other ways. Especially ir. Paul Arendsen, for teaching me that developers and users are two different kind of people. In the end, only the last one counts. Dr. Nikica Petrinic, who has more ideas than time. Dr. Ulf Edlund and Dr. Alastair Johnson, for their material models, and all other partners of the HICAS project, who allowed me to use their data. My friends and colleagues from Delft, ir. Joris Remmers and Dr. Gert Rebel, for their help with stability problems. Noureddine Mansouri, from the Service Informatique, for helping me to get the code running on the Swiss-Tx. Karin Merazzi, for proofreading my report, digging through my English, and for much more. And Michel Daout, for helping me with my French.

ACKNOWLEDGEMENTS

I am grateful to all my friends and colleagues during my stay here. With the risk of forgetting somebody: ir. Niels Hilbrink, for being the computer freak he is; Dr. Frédéric Blom, for sharing my enthusiasm; And all the colleagues of the Institute: Martine Baudin, for all the help with the bureaucracy, Michel, Anne-Isabelle, Bernd, Etienne, Pascal, Sandra, Benny, Yvan, Raymond, Charles, and especially Dr. Branko Glisic for all the coffee breaks, which I needed to get started in the morning.

And last, but certainly not least, my mother, Tiny Volgers-van Veen and my sister, Annelie Volgers, for their support and faith in me, despite being so far away. And to Monica Daout, for basically everything. For your never ending support and for allowing me to spend so much time on my work instead of you.

Curriculum Vitae

Full name	Pieter Theodorus Gerardus Volgers
Date of birth	19-01-1973
Place of birth	Alkmaar, the Netherlands
Nationality	Dutch
Address	Rue du Parc 101
	2300 La Chaux-de-Fonds
	Switzerland

Education:

1985-1991	Petrus Canisius College, Alkmaar
1991-1997	Delft University of Technology
	Department of Aerospace Engineering
	Institute of Structures and Computational Mechanics
	Advisors: E. Riks and J. Arbocz
	Degree: M.Sc. (*ir.*)
1997-	Ecole Polytechnique Fédérale de Lausanne (EPFL)
	Department of Civil Engineering
	Stress Analysis Laboratory
	Advisors: L. Pflug and S. Merazzi

CURRICULUM VITAE

Professional experience:

Sept-Dec. 1995	Trainee at Engineering Systems International (ESI) Paris, France
Feb.-April 1997	Assistant at the Delft University of Technology Department of Aerospace Engineering Delft, the Netherlands
May 1997 -	Ph.D. student at the EPFL Dept. of Civil Engineering Lausanne, Switzerland

Principle area of research:

Computational mechanics, transient explicit analysis
Finite element code development
High-performance computing

Reports, publications:

Contact in finite elements. Master's thesis.
Delft University of Technology, Delft, the Netherlands. 1997

Lagrange-like contact in ETA. Master's thesis.
Delft University of Technology, Delft, the Netherlands. 1997

A review of time-integration methods. Memorandum M-792.
Delft University of Technology, Delft, the Netherlands. 1997

Higher-order contact segment.
Engineering Systems International, Rungis-Cedex, France. 1995

New developments in finite element programming:
Part I, an object-oriented wrapper. (With S. Merazzi)
Paper presented at the 2nd B2000/MemCom Workshop,
Lugano, Switzerland. 1998

New developments in finite element programming:
Part II,introducing parallelism in B2000. (With N. Hilbrink)
Paper presented at the 2nd B2000/MemCom Workshop,
Lugano,Switzerland. 1998

Concurrent development of B2000. (With N. Hilbrink)
Paper presented at the 2nd B2000/MemCom Workshop,
Lugano,Switzerland. 1998

Commodity MPI computers for scientific problem solving.
(With R. Gruber, A. De Vita, M. Stengel and T.M. Tran).
Paper to be presented at the IMACS World Congress,
Lausanne, Switzerland, 2000

High performance computing in computational structural analysis.
(With M. Doreille).
Paper to be published in EPFL Supercomputing Review,
Lausanne, Switzerland, 2000